

USENIX Association

Proceedings of the General Track

2003 USENIX Annual Technical Conference

**June 9–14, 2003
San Antonio, Texas, USA**

Conference Organizers

Program Chair

Brian Noble, *University of Michigan*

Program Committee

Andrea Arpaci-Dusseau, *University of Wisconsin*

Edouard Bugnion, *VMware*

Vinny Cahill, *Trinity College Dublin*

Jason Flinn, *University of Michigan*

Steve Gribble, *University of Washington*

Geoffrey H. Kuenning, *Harvey Mudd College*

Darrell Long, *UCSC*

Patrick McDaniel, *AT&T—Research*

Vern Paxson, *ICSI*

Dave Presotto, *Bell Labs*

Alex Snoeren, *UCSD*

Dawn Song, *CMU*

Marvin Theimer, *Microsoft Research*

Amin Vahdat, *Duke University*

Bennet Yee, *UCSD*

Invited Talks Coordinators

Ted Faber, *USC Information Sciences Institute*

John Ioannidis, *AT&T Labs—Research*

“The Guru Is In” Coordinator

Lee Damon, *University of Washington*

The USENIX Association Staff

External Reviewers

Remzi Arpaci-Dusseau

Anindya Basu

Nathan Burnett

Mark Corner

Landon Cox

Eric Cronin

Ray Cunningham

Jim Dowling

Stephen Farrell

Brian Forney

Kevin Fu

Jun Gao

James Hoe

Minkyong Kim

Yongdae Kim

Steven Osman

Joe Pasquale

Florentina Popovici

Vijayan Prabhakaran

Muthian Sivathanu

Lakshminarayanan Subramanian

Doug Terry

Stefan Weber

Lakshman Yagati

2003 USENIX Annual Technical Conference
General Track
June 9–14, 2003
San Antonio, Texas, USA

Index of Authors	v
Message from the Program Chair	vi

Thursday, June 12, 2003

Administration Magic

Undo for Operators: Building an Undoable E-mail Store	1
<i>Aaron B. Brown and David A. Patterson, University of California, Berkeley</i>	
Role Classification of Hosts Within Enterprise Networks	15
<i>Godfrey Tan, Massachusetts Institute of Technology; Massimiliano Poletto, Mazu Networks; John Guttag and Frans Kaashoek, Massachusetts Institute of Technology</i>	
A Cooperative Internet Backup Scheme	29
<i>Mark Lillibridge, Sameh Elnikety, Andrew Birrell, Mike Burrows, and Michael Isard, HP Systems Research Center</i>	

Power

Currentcy: A Unifying Abstraction for Expressing Energy Management Policies	43
<i>Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat, Duke University</i>	
Design and Implementation of Power-Aware Virtual Memory	57
<i>Hai Huang, Padmanabhan Pillai, and Kang G. Shin, University of Michigan</i>	

Get Virtual

Operating System Support for Virtual Machines	71
<i>Samuel T. King, George W. Dunlap, and Peter M. Chen, University of Michigan</i>	
A Multi-User Virtual Machine	85
<i>Grzegorz Czajkowski and Laurent Daynès, Sun Microsystems Laboratories; Ben Titzer, Purdue University</i>	

Friday, June 13, 2003

Needles and Haystacks

A Logic File System	99
<i>Yoann Padioleau and Olivier Ridoux, IRISA / University of Rennes</i>	
Application-specific Delta-encoding via Resemblance Detection	113
<i>Fred Douglass and Arun Iyengar, IBM T.J. Watson Research Center</i>	
Opportunistic Use of Content Addressable Storage for Distributed File Systems	127
<i>Niraj Tolia, Carnegie Mellon University and Intel Research Pittsburgh; Michael Kozuch, Intel Research Pittsburgh; Mahadev Satyanarayanan, Carnegie Mellon University and Intel Research Pittsburgh; Brad Karp, Intel Research Pittsburgh; Thomas Bressoud, Intel Research Pittsburgh and Denison University; Adrian Perrig, Carnegie Mellon University</i>	

Change Is Constant

System Support for Online Reconfiguration	141
<i>Craig A. N. Soules, Carnegie Mellon University; Jonathan Appavoo and Kevin Hui, University of Toronto; Robert W. Wisniewski and Dilma Da Silva, IBM T.J. Watson Research Center; Gregory R. Ganger, Carnegie Mellon University; Orran Krieger, IBM T.J. Watson Research Center; Michael Stumm, University of Toronto; Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis, IBM T.J. Watson Research Center</i>	
Checkpoints of GUI-based Applications	155
<i>Victor C. Zandy and Barton P. Miller, University of Wisconsin—Madison</i>	
CUP: Controlled Update Propagation in Peer-to-Peer Networks	167
<i>Mema Roussopoulos and Mary Baker, Stanford University</i>	

Security Mechanisms

The Design of the OpenBSD Cryptographic Framework	181
<i>Angelos D. Keromytis, Columbia University; Jason L. Wright and Theo de Raadt, OpenBSD Project</i>	
NCryptfs: A Secure and Convenient Cryptographic File System	197
<i>Charles P. Wright, Michael C. Martino, and Erez Zadok, Stony Brook University</i>	
A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks	211
<i>Manish Prasad and Tzi-cker Chiueh, Stony Brook University</i>	

Saturday, June 14, 2003

Fast Servers

Kernel Support for Faster Web Proxies	225
<i>Marcel-Cătălin Roșu and Daniela Roșu, IBM T.J. Watson Research Center</i>	
Multiprocessor Support for Event-Driven Programs	239
<i>Nickolai Zeldovich, Stanford University; Alexander Yip, Frank Dabek, and Robert T. Morris, Massachusetts Institute of Technology; David Mazières, New York University; Frans Kaashoek, Massachusetts Institute of Technology</i>	

Big Data

Seneca: Remote Mirroring Done Write	253
<i>Minwen Ji, Alistair Veitch, and John Wilkes, HP Laboratories</i>	
Eviction-based Cache Placement for Storage Caches	269
<i>Zhifeng Chen and Yuanyuan Zhou, University of Illinois at Urbana-Champaign; Kai Li, Princeton University</i>	
Fast, Scalable Disk Imaging with Frisbee	283
<i>Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb, University of Utah</i>	

I/O Guessing Games

Robust, Portable I/O Scheduling with the Disk Mimic	297
<i>Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison</i>	
Controlling Your PLACE in the File System with Gray-box Techniques	311
<i>James A. Nugent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison</i>	
Operating System I/O Speculation: How Two Invocations Are Faster Than One	325
<i>Keir Fraser and Fay Chang, Compaq Systems Research Center</i>	

Index of Authors

Appavoo, Jonathan	141	Martino, Michael C.	197
Arpaci-Dusseau, Andrea C.	297, 311	Mazières, David	239
Arpaci-Dusseau, Remzi H.	297, 311	Miller, Barton P.	155
Auslander, Marc	141	Morris, Robert T.	239
Baker, Mary	167	Nugent, James A.	311
Barb, Chad	283	Ostrowski, Michal	141
Birrell, Andrew	29	Padioleau, Yoann	99
Bressoud, Thomas	127	Patterson, David A.	1
Brown, Aaron B.	1	Perrig, Adrian	127
Burrows, Mike	29	Pillai, Padmanabhan	57
Chang, Fay	325	Poletto, Massimiliano	15
Chen, Peter M.	71	Popovici, Florentina I.	297
Chen, Zhifeng	269	Prasad, Manish	211
Chiuueh, Tzi-cker	211	Ricci, Robert	283
Czajkowski, Grzegorz	85	Ridoux, Olivier	99
Dabek, Frank	239	Rosenburg, Bryan	141
Da Silva, Dilma	141	Roşu, Daniela	225
Daynès, Laurent	85	Roşu, Marcel-Cătălin	225
de Raadt, Theo	181	Roussopoulos, Mema	167
Douglis, Fred	113	Satyanarayanan, Mahadev	127
Dunlap, George W.	71	Shin, Kang G.	57
Ellis, Carla S.	43	Soules, Craig A. N.	141
Elnikety, Sameh	29	Stoller, Leigh	283
Fraser, Keir	325	Stumm, Michael	141
Ganger, Gregory R.	141	Tan, Godfrey	15
Guttag, John	15	Titzer, Ben	85
Hibler, Mike	283	Tolia, Niraj	127
Huang, Hai	57	Vahdat, Amin	43
Hui, Kevin	141	Veitch, Alistair	253
Isard, Michael	29	Wilkes, John	253
Iyengar, Arun	113	Wisniewski, Robert W.	141
Ji, Minwen	253	Wright, Charles P.	197
Kaashoek, Frans	15, 239	Wright, Jason L.	181
Karp, Brad	127	Xenidis, Jimi	141
Keromytis, Angelos D.	181	Yip, Alexander	239
King, Samuel T.	71	Zadok, Erez	197
Kozuch, Michael	127	Zandy, Victor C.	155
Krieger, Orran	141	Zeldovich, Nickolai	239
Lebeck, Alvin R.	43	Zeng, Heng	43
Lepreau, Jay	283	Zhou, Yuanyuan	269
Li, Kai	269		
Lillibridge, Mark	29		

Message from the Program Chair

Dear Colleagues:

Every year, the USENIX Annual Technical Conference brings together the best and brightest in the field, highlighting the best and most current technical work. I'm pleased to say that this year will continue in this excellent tradition.

This year, the General Track received 104 submissions. Each paper was rigorously reviewed by three separate program committee members for technical content and quality. We then met as a group in Ann Arbor, Michigan, on a very cold January day to discuss 61 of the submissions. The committee was impressed with the breadth and number of high-quality papers, making final selections difficult. At the end of the meeting, we had selected 24 outstanding papers, and believe you will enjoy and learn from the technical program.

In what may be a first-ever occurrence, every program committee member turned in their reviews before my "optimistic" deadline for receiving them. I'm embarrassed to admit that my reviews were among the last ones filed! The USENIX General Track review cycle is a tight one, not considering the winter holidays in its midst. I found the quality and depth of the reviews to be excellent, and I could not have asked more from any of the PC members. Thank you all!

Those who have worked with the USENIX staff before know the lengths to which they will go for the conference. In particular, thanks go to Peter Honeyman, my board liaison, colleague, and supplier of candy to the program committee. Last, and most importantly, thanks are due to Jane-Ellen Long and Ellie Young for keeping me on schedule and providing the support that makes USENIX conferences "must-attend" events.

So enjoy the 2003 edition of the USENIX Annual Technical Conference.

Brian Noble, *University of Michigan*
Program Chair

Undo for Operators: Building an Undoable E-mail Store

Aaron B. Brown and David A. Patterson

University of California, Berkeley, EECS Computer Science Division

387 Soda Hall #1776, Berkeley, CA, 94720-1776, USA

{abrown,patterson}@cs.berkeley.edu

Abstract

System operators play a critical role in maintaining server dependability yet lack powerful tools to help them do so. To help address this unfulfilled need, we describe *Operator Undo*, a tool that provides a forgiving operations environment by allowing operators to recover from their own mistakes, from unanticipated software problems, and from intentional or accidental data corruption. *Operator Undo* starts by intercepting and logging user interactions with a network service before they enter the system, creating a record of user intent. During an undo cycle, all system hard state is physically rewound, allowing the operator to perform arbitrary repairs; after repairs are complete, lost user data is reintegrated into the repaired system by replaying the logged user interactions while tracking and compensating for any resulting externally-visible inconsistencies. We describe the design and implementation of an application-neutral framework for *Operator Undo*, and detail the process by which we instantiated the framework in the form of an undo-capable e-mail store supporting SMTP mail delivery and IMAP mail retrieval. Our proof-of-concept e-mail implementation imposes only a small performance overhead, and can store days or weeks of recovery log on a single disk.

1 Introduction

Dependability is one of the greatest challenges facing the designers and implementors of today's enterprise and Internet services, yet even as industry and researchers strive to build more dependable hardware and software [11] [22], dependability today is still largely delivered or lost by the human beings who operate and administer service installations. Human operators are entrusted with the power and responsibility to configure service systems and keep them running despite frequent upgrades and unexpected failure, but, like any humans, they are prone to human error and thus can themselves be a significant impediment to dependability [4].

Despite their critical role in maintaining dependability, system operators are confronted with an unforgiving environment offering little support for carrying out that role. Configuration, upgrades, diagnosis, repair, and recovery at each layer of the system are typically performed with an ad-hoc collection of independent tools. Mistakes can have catastrophic consequences, including loss or corruption of user data, and thus there is little ability to explore and experiment with different potential solutions. Furthermore, in today's complex, tightly-coupled, rapidly-changing systems, operators face precisely those dependability problems that are most likely to result in mistakes: unfamiliar situations with complex interactions and underspecified symptoms [24]. It should come as no surprise, then, that human operator error is pegged as the root cause of roughly 20% to 50% of system outages [10] [18] [20].

Consider, as an example, what problems an operator might face in the day-to-day administration of a corporate or ISP e-mail store. She might be asked to add a new virtual host to the system's configuration; what if, upon doing so, she inadvertently alters the configuration so that mail to existing accounts starts to bounce? Maybe she knows what she did wrong and can go fix it, but even if so, e-mail may be lost in the interim. And if the problem is harder to track down, the system could operate improperly for hours or days, much as happened with Microsoft's DNS servers during a widely-publicized 24-hour outage that was ultimately tracked down to an inadvertent operator configuration mistake [14].

Or what if our administrator is asked to set up a spam filter on the e-mail store, and she configures it incorrectly the first time around? Again, mail could be lost for a lengthy period while the problem is tracked down and resolved. Or consider the case where the operator installs a software upgrade/patch only to find that it performs poorly—or worse, corrupts data—when deployed at full scale. Maybe the system could be restored from backup, but what about the intervening data that are then lost?

Now, imagine that our operator has a tool available to her that provides a system-wide version of the Undo functionality that we have all grown accustomed to in our word processors and productivity applications. In each of the above scenarios, she could use Undo to restore the system back in time to a point before things went wrong. She could then make repairs, retry the pro-

cedure that went wrong the first time, and, with an appropriately-designed Undo system, roll the system forward again, replaying all of the e-mail deliveries and user mailbox operations that were lost or handled incorrectly the first time around.

Unfortunately, this notion of undo, so common in today's productivity applications, is virtually unheard of in the administration and operations environment. We are trying to change that through our research. In this paper, we present the design and implementation of a proof-of-concept Undo system for network-delivered service applications. Our first target application for Undo is an e-mail store system that receives mail via SMTP and provides retrieval access via IMAP. We chose e-mail because it is a widely-deployed, increasingly-mission-critical service; studies report that up to 45% of critical business information is stored in e-mail [19], and that loss of e-mail access can result in up to a 35% decrease in worker productivity [21]. However, despite our initial focus on e-mail, much of our undo system is designed to be service-neutral, and should apply directly to other systems providing network-delivered services.

In the remainder of this paper, we first present an overview of our model for Operator Undo in Section 2. In Section 3, we explore a fleshed-out design for a service-neutral undo manager that implements our undo model. Section 4 describes the integration of the generic undo design with the specific application of an e-mail message store. We analyze the feasibility of providing Operator Undo for e-mail in terms of resource and time overhead in Section 5, then wrap up with related work in Section 6 and future work and conclusions in Section 7.

2 The Three R's Model of Operator Undo

An undo facility is the ideal counterpoint to the dependability problems faced by system operators. It provides a forgiving environment by allowing operators to recover from their mistakes, to handle unexpected situations by exploring and experimenting with alternative solutions to problems, and by reducing the stress and cognitive strain that arise when every action may be catastrophic. A further benefit is that an undo system can be used by operators as a recovery mechanism for non-human-instigated problems. Just as the system can be "undone" to remove the effects of an operator error, it can be wound back to cancel out corruption due to software bugs, to reverse unanticipated effects of a patch or upgrade, and perhaps even to remove the damage done by a malicious hacker or virus attack.

In previous work, we outlined a model for Operator Undo that provides these benefits and sketched the beginnings of a design for a service-neutral undo man-

ager [3]. We recap that work here then proceed to flesh it out into a practical design and a real implementation.

Our model for Operator Undo is based on three fundamental steps that we refer to as the "Three R's": **Rewind**, **Repair**, and **Replay**. In the Rewind step, all system state (OS through application) is physically rolled back in time to a point before any catastrophic damage occurred. In the Repair step, the operator alters the rolled-back system to prevent the problem from reoccurring. Note that repairs are not constrained by our model and can consist of arbitrary changes to the system or to the rewound part of the timeline. Finally, in the Replay step, the repaired system is rolled forward to the present by replaying portions of the previously-rewound timeline in the context of the repaired system.

The essence of Three-R's Undo, and the property that distinguishes it from more traditional approaches like backup/restore, is that it preserves the system timeline: it restores lost updates and incoming data on replay in a manner that retains their intent and not the (possibly incorrect) results of their original processing. In all the scenarios discussed in Section 1, Three-R's Undo would have restored lost incoming mail and user mailbox updates, re-executing them on the repaired system where they could be processed correctly. It is this restorative ability that gives Three-R's Undo its power as a tool for the system operator.

2.1 Three-R's design decisions

There are a few essential design decisions captured in the Three-R's undo model as we have described it. First is the choice to perform Rewind physically and Replay logically. In this approach, "undo" is implemented by the single operation of restoring a previous snapshot of a system's hard state, and "redo" is implemented by re-executing a recorded sequence of user-level operations. Physical rewind provides the greatest flexibility in recovering from problems because the undo system makes no assumptions about the semantics of state or the possible corruptions it might encounter. Alternate rewind schemes involving logical rollback would require such knowledge, and risk the possibility of corrupt state escaping rewind due to bugs or unanticipated failure modes. Furthermore, by rolling back *all* state, we do not need to worry about corrupt state escaping the rewind roll-back and persisting to cause problems during replay.

In contrast, logical replay is mandatory if the undo system is to integrate changes made during repair with the system's original timeline. Whereas physical replay would obliterate any fixes made during repair as it rolled the original, corrupted version of state forward over the repairs, logical replay preserves the intent of user operations without reference to the original cor-

rupted state and while still respecting repairs. While logical replay threatens to increase the undo system's complexity and hence the possibility of dependability-affecting bugs, we construct the undo system so that the code used for replay is exercised as part of normal system operation, thereby flushing out any bugs before the replay code must be relied upon during an emergency.

Another key design decision for the Three-R's is that Repair be as unconstrained as possible to allow the operator full flexibility in designing solutions to repair system problems. Often the most confounding and error-prone problems are the ones that have never been seen before. Constraining the Three-R's undo system to a well known set of actions would render it ineffective in exactly those scenarios where it is needed the most.

Finally, note that these design decisions, particularly the choices of physical rewind and unconstrained repair, reflect a fault model that makes minimal assumptions about the correctness of the undoable application service. While this fault model may limit the ability to formally analyze the undo process, it is the key to practical recovery from problems that have altered the proper operation of the system in unknown ways. These are exactly the classes of problems a system operator is likely to encounter when the system is subject to erroneous operator intervention, software bugs, and faulty patches or upgrades.

2.2 Challenges in the Three-R's model

Given the design decisions we have made, there are two key challenges in the Three-R's model. The first is timeline management: to provide the time-travel-like behavior of the Three-R's, an operator-undo system must record the system's timeline so that it can be edited during Repair and re-executed during Replay. In doing so, the undo system must accurately capture the intent of all state changes made by the system's end users in such a way that they can be later replayed while still respecting the alterations made to the system during Repair. Furthermore, the recorded timeline must be causally consistent with the actual execution of the system: all non-commuting operations must be recorded in the same order they were originally executed.

The second key challenge in the Three-R's model is to keep the system consistent from the point of view of an external observer. As in the time travel paradoxes in popular science fiction stories, Three-R's undo can result in a system that appears inconsistent across time to an external observer. This occurs when alterations made during Repair cause state that had already been seen by the observer before Rewind to take on new values during Replay. For example, a repair that affects an e-mail server's spam filter could cause previously viewed e-mail messages to change or be removed, caus-

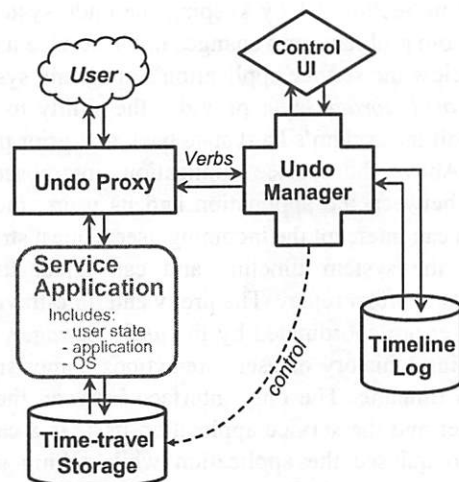


Figure 1: Undo system architecture. The heart of the undo system is the undo manager, which coordinates the system timeline. The proxy and time-travel storage layer wrap the service application, capturing and replaying user requests from above and providing physical rewind from below.

ing the system to appear inconsistent to the observer of those e-mail messages. Note that inconsistencies in state *not* already seen by an external observer are acceptable—even desirable—as they represent the positive effects of repairs; it is only when the inconsistencies are in previously-viewed state that they must be managed.

3 Design of a Generic Undo System

While our discussion in this paper focuses on providing Operator Undo for an e-mail environment, a primary goal while developing the architecture and implementation of Operator Undo was to produce a tool that would work with as many enterprise- and Internet-service applications as possible. While some parts of an operator undo implementation are necessarily service-specific (the model of acceptable external consistency, for example), much of the mechanism can be built to be reusable and service-neutral. This is an important consideration for a system targeted at increasing dependability, as any complexity added by the undo system increases the likelihood of dependability problems due to software bugs. If the complicated undo mechanisms are built *once* in a generic manner and then reused as undo is added to each new service, bugs in the undo mechanisms will get flushed out quickly, resulting in a more robust system than if the undo mechanisms were built anew each time.

3.1 Undo system architecture

To this end, our undo system design follows the structure illustrated in Figure 1. The service application—such as an e-mail store server—and its hosting operating system are left virtually unmodified; the undo system interposes itself both above and below the service. This wrapper-based approach supports the fault model dis-

cussed in Section 2.1 by keeping the undo system isolated from problems and changes in the service itself.

Below the service application's operating system, a *time-travel storage* layer provides the ability to physically roll the system's hard state back to a prior point in time. Above the service application, interposed as a proxy between the application and its users, the undo system can intercept the incoming user request stream to record the system timeline and can inject its own requests to effect replay. The proxy and time-travel storage layer are coordinated by the *undo manager*, which maintains a history of user interactions comprising the system timeline. The only interface between the undo manager and the service application itself is a callback used to quiesce the application while taking storage checkpoints or rewinding.

For simplicity, we make a few assumptions about the service application. We assume that it includes internal recovery mechanisms that allow it to reconstruct its internal state from a storage checkpoint, and that it flushes permanent state changes resulting from user interactions to stable storage before responding to the user. These assumptions allow us to coordinate the time-travel storage and timeline log without further hooks into the application; having such hooks would allow us to relax the assumptions at the cost of tighter integration.

The use of a proxy-based approach (rather than an approach where the service and undo manager interact directly) biases our implementation toward services in which users interact with the service via a narrow, well-defined interface, or protocol. Certainly, an e-mail service fits this model, with its use of protocols like IMAP and SMTP. And most Internet services are based on open protocols, while many enterprise services are being developed on middleware that uses XML/SOAP-based protocols for communication. In cases where the user accesses the service via a web front-end and not via a well-defined protocol, the Undo proxy can be inserted at the interface between the web tier and the application/middleware tier.

Despite limiting the range of services that can be easily adapted to support undo, the proxy-based approach has significant benefits. First, for applications with standard protocols like e-mail, a protocol-specific proxy can be developed once and then reused across service implementations, again helping to address the fear that the proxy may introduce extra complexity and hence bugs. Along the same lines, using a protocol-specific proxy rather than integrating undo functionality into the application allows repairs to consist of sweeping changes to the system—such as upgrading or replacing the OS or application—while still allowing replay, as long as the protocols themselves have not changed.

Finally, notice that the service in Figure 1 is depicted as a single monolithic block, with a single entry point for user requests. In this simple version of the undo system design, the entire service is rolled back and forward in time during the Three-R's undo cycle. While this is how we have developed our initial proof-of-concept implementation, we believe the architecture can be extended to support a distributed proxy and clustered service architecture. The extension is straightforward in the case where each service node handles an independent subset of the system's users, but may require the use of more sophisticated techniques from the distributed checkpointing and dependency management domains when shared state is involved.

3.2 Verbs: the undo manager interface

The only service-specific component in the architecture of Figure 1 is the proxy that interposes on the service's user-request stream. Clearly, the proxy itself will be application-specific, as it must understand the protocols it is proxying. But the proxy communicates with the undo manager, a component that itself has no knowledge of the service or its semantics, so it must translate user requests into and out of a form that can be handled generically by the undo manager. At the same time, the undo manager must be able to reason about those translated requests in order to address the challenges of timeline management and external consistency discussed above in Section 2.2.

The answer to this seemingly contradictory set of requirements lies in *verbs*, the fundamental construct used to represent events in the system timeline. A verb is an encapsulation of a user interaction with the system—a record of an event that causes state in the service to be changed or *externalized* (exposed to an external observer). To achieve the separation of application-specific proxy and application-independent undo manager, verbs are transparent to the proxy while semi-opaque to the undo manager: a verb contains all the application-specific information needed to execute or re-execute its corresponding user interaction, but to the undo manager appears as only a generic data type with interfaces exposing just enough information to manage the verb's recording and execution.

To decouple the record of user interactions from the specific behavior of the application service in processing those interactions, verbs record the *intent* of user interactions as expressed at the protocol level, rather than recording the effects of those interactions on state or the contents of state itself. For example, when a user deletes an e-mail message, a verb is created that specifies the deletion intent and a name that uniquely identifies the target message. As part of recording intent, verbs must capture any system context required to spec-

ify the behavior of the verb; for example, converting times specified relative to the present into absolute times. In this sense, the task of defining verbs involves similar processes as the task of defining conformance wrappers for Byzantine replication, as defined by Rodrigues et al [25]. Note that designing verbs to capture intent achieves the critical goal identified in Section 2.1 of allowing the Undo system to tolerate faulty application behavior during normal (non-Undo) operation, and makes it possible to replay verbs in the context of a repaired system.

Verbs are used in the undo system during all phases of operation. During normal operation of the service, the proxy intercepts end-user interactions that change or externalize state, packages them into verbs, and ships them to the undo manager for processing. The undo manager uses the verb interfaces to generate a causally consistent ordering of the verbs it receives, sends the verbs back to the proxy for execution on the service system, and records the sequence of executed verbs in an on disk log. This verb log forms the recorded timeline of the system. During the Repair phase, the timeline may be edited to remove, replace, or add verbs, or may be left unaltered if repairs are done directly to the service itself. During the Replay phase, the undo manager attempts to re-execute the appropriate portion of the timeline by shipping logged verbs back to the proxy for execution on the service system. As it does this, it uses verb interfaces to determine if external inconsistencies are being created, and if so, invokes other verb interfaces to perform application-specific compensation. Note that the same code is used to re-execute the verbs during replay as to execute them during normal operation, helping to ensure that the replay code is bug-free and dependable. The flow of verbs during normal operation and during Replay are illustrated in Figure 2.

3.2.1 Verb interfaces

Section 2.2 introduced two key challenges in building Three-R's-undo: timeline management and external inconsistency management. To address these challenges, verbs define a set of interfaces that provide the undo manager with a window into the application-specific semantics of verb execution, thus exposing enough information to allow the undo manager to carry out its management tasks. These interfaces fall into two groups, discussed in turn below.

Sequencing interfaces. The first set of verb interfaces is used to generate a recorded timeline that is consistent with the actual execution of the system, thereby addressing the challenge of timeline management. It consists of three procedures that all verbs must define: a commutativity test, an independence test, and a preferred-order-

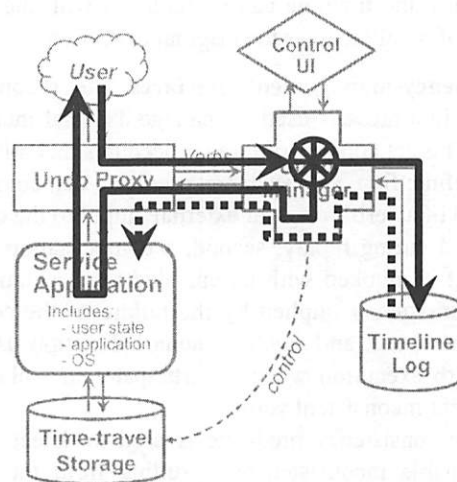


Figure 2: Illustration of verb flow. During normal operation, the verb flow follows the solid black arrows, with verbs created in the proxy and looped through the undo manager for scheduling and logging. During replay, verb flow follows the heavy dashed arrow, with verbs being reconstructed from the timeline log and re-executed via the proxy.

ing test. All three tests take another verb as an argument; the first tests if the two verbs produce the same results regardless of execution order, the second tests if the verbs can be safely executed in parallel, and the third returns a preferred execution ordering of the two verbs in the case where they do not commute. Note that these tests are similar to those defined by actions in the IceCube optimistic replication system, although in that case they are used for log merging and reordering rather than execution control [23].

The sequencing tests are used by the undo manager to generate a consistent timeline log when faced with multiple verbs arriving concurrently from multiple users. Because verbs are generated as they arrive at the proxy, whereas their corresponding user interactions are only sequenced for execution in the service application, it is possible that the proxy sees overlapping interactions arrive in a different order than that in which they are eventually executed. Using the sequencing tests, however, the undo manager can guarantee that it sees the same execution ordering as will be chosen by the service application: it can simply stall each incoming verb until all in-flight non-commuting/non-independent verbs have completed execution, using a scoreboard-like data structure to manage the out-of-order execution.

While this approach does involve some serialization of arriving user interactions, it executes as many as possible in parallel, serializing only when there is a non-commutative dependency between concurrently-arriving interactions. It produces a timeline that, when replayed serially, will result in a system state consistent with that produced by the original execution. Furthermore, using the same independence and commutativity

properties, the timeline can be replayed with the same degree of parallelism as the original execution.

Consistency-management interfaces. The second set of verb interfaces is used to manage external inconsistency. This set consists of three procedures that all verbs must define: first, a consistency predicate that compares a record of a verb's original external output to the output produced during replay; second, a compensation function that is invoked with an encoded representation of the inconsistency implied by the failure of the consistency predicate; and finally a squash function used to alter verb execution when it participates in a chain of dependent inconsistent verbs.

The consistency predicate is used to detect externally visible inconsistencies resulting from the undo cycle. Verbs that externalize output record a copy (or hash) of that output when they are originally executed and again during replay. The consistency predicate is applied by the undo manager after the externalizing verb is replayed, and compares the two sets of output to determine if they are acceptably consistent; this test may be simply an equality test, or may be more sophisticated if the application allows relaxed external consistency.

If the consistency predicate fails, the undo manager invokes the second interface, the compensation procedure, which can take whatever application-defined action is necessary to handle the inconsistency. Compensation may consist of ignoring the inconsistency, performing some action to mitigate it (such as creating a missing piece of state), or explaining the inconsistency to the user, among other possibilities.

One final concern involves handling user-induced dependencies between verbs that produce external inconsistencies and later verbs in the timeline. For example, in an e-mail system, a user might choose to delete a message based on reading its content. If during a later undo cycle that (externalized) content is changed, the user's decision to delete the message might be invalid. Given the limited amount of insight into user intent available to the undo system, a conservative approach to handling such scenarios is necessary. The approach we chose is to have the undo manager invoke the third interface, the squash procedure, on all later verbs that do not commute with a verb that externalizes state. Squashing, like compensation, is application-defined, but typically consists of cancelling the verb's original action, informing the user, and leaving it up to them to reconstruct their original intent. Typically, only verbs that destroy or overwrite state will choose to alter their execution when squashed. This policy minimizes the amount of user cleanup needed should a long chain of dependent verbs appear, while ensuring that no potentially-valuable state is lost.

Discussion. The verb interfaces for sequencing and external consistency management bear more than a passing resemblance to similar interfaces used to manage consistency in weakly-connected optimistically-replicated storage systems such as Bayou [30], IceCube [12], and Coda [28]. The similarity is not surprising: the problem of replaying user verbs after the repair phase of Three-R's Undo is somewhat analogous to the task of using an operation log to update an out-of-sync replica in an optimistically-replicated storage system, and our approach is modeled after approaches in that domain.

The key difference in the domain of Three-R's undo is that, unlike in replica systems, not every inconsistency matters—in fact, most inconsistencies that arise are likely due to the positive impact of repairs, representing earlier misbehaviors that are now corrected, and should be silently preserved. This insight motivated the choice of only testing for consistency of external output, rather than using preconditions to test every verb for inconsistency before executing it, as is done in systems like Bayou [30]. We do share Bayou's notion of application-defined compensations; they are just applied in different situations in our system, namely only at the point at which the effects of an inconsistency cross the external boundary of the system.

Another key difference from replica systems is the use of *post-execution* consistency checks in Undo, rather than pre-execution checks. The reason for this is that, in Undo, inconsistencies arise only during replay, not normal operation, and thus can be safely detected after the fact. The built-in rewind functionality can be used to unwind execution to properly compensate for a detected inconsistency, if necessary. Using only post-execution checks simplifies our design, as it is much easier to compare a verb's actual output for consistency than to predict whether its inputs will produce a consistent result, especially given our lack of assumptions about the service's correctness.

3.2.2 Handling failed verbs

Special handling is required when verb execution fails during normal execution. If the verb's corresponding operation reports its status back to the end user synchronously, we do not record the verb as part of the system timeline, and thus the corresponding operation will not be retried upon replay. While this may seem counter to the goals of an Undo system, the problem with recording and later replaying synchronous failed verbs is that the subsequent timeline—the user's choice of future requests—is informed by the failure, and may not make sense if the failure is converted to a success. For example, if the user attempts to create a mail folder with an illegal name, he or she will see the failure and will likely go and try to create another folder using a valid name. If

Verb	Protocol	Changes state?	Externalizes state?	Async?	Description
Deliver	SMTP	✓		✓	Delivers a message to the mail store via SMTP
Append	IMAP	✓			Appends a message to a specific IMAP folder
Fetch	IMAP	✓	✓		Retrieves headers, messages, or flags from a folder
Store	IMAP	✓	✓		Sets flags on a message (<i>e.g.</i> , Seen, Deleted)
Copy	IMAP	✓			Copies messages to another IMAP folder
List	IMAP		✓		Lists extant IMAP folders
Status	IMAP				Reports folder status (<i>e.g.</i> , message count)
Select	IMAP				Opens an IMAP folder for use by later commands
Expunge	IMAP	✓	✓		Purges all messages with Deleted flag set from a folder
Close	IMAP	✓			Performs a silent expunge then deselects the folder
Create	IMAP	✓			Creates a new IMAP folder or hierarchy
Rename	IMAP	✓			Renames an IMAP folder or hierarchy
Delete	IMAP	✓			Deletes an IMAP folder or hierarchy

Table 1: Verbs defined for undoable e-mail store

during a later undo cycle the system is changed to accept the illegal name, it makes no sense to create the original illegally-named folder, as the user has already reacted to that failure and altered course accordingly.

On the other hand, if the verb that fails during normal operation corresponds to an asynchronous operation, we have a great deal more flexibility. By delaying the reporting of failure to the user, we create a window during which it is possible to invoke undo, fix the problem that caused the verb to be rejected, and then successfully re-execute the verb, without affecting the user's future choice of timeline. This allows us to handle situations such as when an e-mail system is misconfigured to reject e-mail: by delaying bounces or making them tentative, we can provide a window of time during which the configuration can be fixed transparently to the senders of the originally-rejected mail (we discuss this particular scenario further in Section 4.1.2). Of course, once a failed asynchronous verb's results have been reported, we must treat it like a synchronous verb and refuse to replay it. To make this scheme work, we require that verbs identify themselves as synchronous or asynchronous, and, if asynchronous, specify the time window between execution and status visibility.

The other context in which failed verbs become an issue is during replay, when an originally-successful verb fails on re-execution. This case is much simpler than the ones just discussed; we treat the verb's failure status as simply another piece of externalized state, and apply the same mechanisms described in Section 3.2.1 for handling inconsistency in externalized data.

4 Implementing Undo in an E-mail Store

Now we turn to our implementation of the just-described architecture for an e-mail store service, which we define as a service representing a leaf node in the

global e-mail network, receiving e-mail for its own local users via SMTP [13] and making it available for reading via IMAP [5]. We focus on what we had to do to adapt the generic architecture for e-mail and the interesting issues we encountered while realizing the implementation. We will not dwell on components whose implementation required just a straightforward translation of the above design into code, like the undo manager itself.

Our implementation, written in Java to leverage its dependability-increasing language features, wraps an unmodified, existing e-mail store. It comprises about 25K lines of Java code, split about evenly between the generic and e-mail-specific components. The e-mail specific part took about two man-months to implement.

4.1 Verbs for e-mail

We defined a set of 13 verbs for our undoable e-mail store that together capture the important state-altering or state-externalizing interactions in the IMAP and SMTP protocols; they are listed in Table 1.

Note that some of the verbs listed, such as Select, do not alter or externalize state but are defined so that they can be properly sequenced by the undo manager as described in Section 3.2.1. Notice also that all of the IMAP verbs are synchronous, as expected given that IMAP is a request-response protocol, whereas the SMTP Deliver verb is asynchronous, reflecting the asynchronous nature of mail transport. Finally, note that while our set of verbs covers only the most commonly-used e-mail functionality for simplicity, it could be extended to encompass some of the more obscure IMAP functionality (such as subscription management) and to capture basic administrative tasks that are performed through interfaces outside of IMAP or SMTP, notably account creation, deletion, and configuration.

Each e-mail verb is implemented as a Java class that implements a common `Verb` interface; the `Verb` interface is defined by the undo manager and declares an API that is a straightforward mapping of the routines described in Section 3.2.1 into Java function declarations. All verbs contain a *tag*, a container data structure that wraps the information needed to execute the verb and to check its external consistency, including a record of whether its execution succeeded or failed. Other than the verb's Java type, the tag is the only part of the verb that is recorded as part of the system's timeline, so it has to be sufficient to reconstruct the verb during replay.

4.1.1 Managing context

One of the challenges in defining verbs for existing protocols like IMAP and SMTP is being able to capture all the necessary context needed to successfully replay the verbs. For SMTP, this is straightforward: we simply capture the parameters passed in to each SMTP command and store them in the corresponding verb's tag.

Applying the same approach to IMAP proved more problematic. In IMAP, operations name state (folders and messages) using names that are only meaningful in a particular system context. In particular, messages are named either by sequence numbers that change any time a message is added to or removed from a folder, or by so-called "unique" IDs that are only unique to a particular instance of a folder and can be unilaterally invalidated at any point by the IMAP server. Similarly, folders are named by hierarchical names that change any time a folder's parent is renamed.

In order to be able to replay IMAP verbs in situations where repairs have changed the system context, we needed to ensure that the verbs specified only absolute names that would still be meaningful after repair. To accomplish this, we defined the notion of an *UndoID*, a time-invariant name independent of system context and capable of being translated into a current IMAP name for verb execution; the proxy is responsible for converting *UndoIDs* to and from IMAP names based on the current system context.

In the case of e-mail messages, *UndoIDs* are allocated and inserted into a reserved message header field whenever a message is injected into the mail store (either by an SMTP Deliver verb or an IMAP Append verb). Then, when creating a verb out of any IMAP command that referred to specific messages, we translate the IMAP names into *UndoIDs* by fetching the *UndoID* directly from the message headers. To replay a verb, we translate the *UndoIDs* back to IMAP names by scanning the folder once to retrieve the *UndoID*-to-IMAP-ID translations, which are then cached for the duration of the Replay cycle.

The case of folders is more difficult, since there is no place to embed the *UndoID* in the folder name. To solve this problem, we built a module that we call the *UIDFactory*. It maintains a mapping of *UndoIDs* to names in a persistent BerkeleyDB database, and is synchronized with the undo manager so that names are invalidated and restored appropriately when the system is rolled back and forward in time. The *UIDFactory* is designed to be general and reusable, treating names as opaque Java objects. In our e-mail system, the *UIDFactory* maps *UndoIDs* to IMAP folder names consisting of an ASCII string for the name of the folder plus the *UndoID* of the folder's parent.

4.1.2 External consistency model

The first step in implementing the inconsistency management architecture of Section 3.2 for e-mail is to define a model of acceptable external consistency. In doing so, we make a distinction between the transport (SMTP) and retrieval (IMAP) phases of e-mail processing. The transport phase allows for a much more relaxed consistency model than the retrieval phase, since even without an undo system, e-mail transport can result in delayed or out-of-order messages. On the retrieval side, consistency has to be stronger, as users are not used to seeing messages or folders change, appear, or disappear from their Inbox without warning. However, even though users are not used to such inconsistencies, we believe that they will accept them if they are sufficiently explained—there is already evidence for this in the numerous mail filters that delete or alter suspected virus- or spam-containing messages, replacing the original message with an explanatory placeholder.

On the retrieval side, we define externalized state to include the output of message fetches (*i.e.*, the text of e-mail messages, including attachments), the output of message list commands (*i.e.*, the standard e-mail headers, including To, From, Subject, and Cc, but not Date), the output of folder list commands (*i.e.*, the currently extant folders in a user's mail store), and the execution status of any state-altering interactive IMAP commands. We declare this state to be inconsistent upon replay if any objects (messages or folders) that were visible originally are missing or altered on replay or if state-altering commands fail. We discount ordering differences and ignore newly-found objects that were not present during original execution, as such discrepancies are typically masked by sorting in the user's e-mail client.

For the most part, we compensate for detected external inconsistencies on the retrieval side by inserting explanatory messages into the user's mailbox, apologizing for the inconsistencies, explaining what they are, and saying why they were necessary. When new folders or messages are being added to the system, we

can be more clever. For example, when the target folder for a message Append verb is found to be missing, we compensate by creating that folder in a special Lost&Found folder in the user's mail store with the same UndoID as the missing folder. By reusing the UndoID, further replayed operations directed at the missing folder go to its newly created surrogate.

From the undo system's perspective, the transport side of e-mail consists only of the SMTP Deliver verb. As SMTP delivery is asynchronous and only reports back to the sender on failure, the one tricky consistency problem is when a formerly-failed message delivery succeeds on replay. If the failure has already been reported to the sender via a standard bounce message, the undo system must not deliver the message during replay, as it does not know what actions the sender took in response to the bounce. However, by delaying the delivery of the final bounce message (typically, by 4 hours), we create a window in which the operator can use Undo to fix mistakes that would cause mail to bounce erroneously. To avoid aggravating users who are used to getting instant feedback on misaddressed e-mails, a failed SMTP Deliver verb sends an informational "bounce" immediately, informing the original sender that the delivery attempt failed but will be retried for the (typically 4-hour) length of the undo window. Note that any inconsistencies in message content, handling, or recipients are not externalized until message retrieval, so they are handled at that point.

All of the consistency checks and compensations described above are implemented through the Verb API in conjunction with the verb tag. When a verb is executed, it updates the tag with a record of the externalized state and the verb's execution status. To reduce the amount of data that must be stored in the timeline, most e-mail verbs record only hashes of their output in the verb tag. Verbs define check routines that compare the tags from the original and replay executions, and compensation routines that perform verb-specific compensations such as generating explanatory messages.

4.1.3 Commutativity and independence

As required by the Verb interface, all of our e-mail verbs define methods for determining if one verb commutes with or is independent of another. These determinations are made by examining the verb types and the contents of their tags, and can often be made simply. For example, any two SMTP verbs are independent of and commute with one another, since message ordering is not considered in our consistency model for e-mail. Similarly, any two IMAP verbs belonging to different IMAP users are independent and commutative, since each user's mail store is independent of all others. To facilitate this determination, IMAP verbs store their

associated username in the verb tag. SMTP and IMAP verbs commute with one another unless the IMAP verb is a Fetch for a user's Inbox; in this case we conservatively mark the verbs as non-commutative since, due to the existence of aliases and mailing lists, it is impossible to determine from the proxy level who is the actual recipient of an arriving SMTP message.

Given these rules, the only remaining case is of two IMAP verbs for the same user. Here, the tests have to be more extensive, examining the input parameters in the two verbs' tags to determine if they commute. For example, an Expunge and a Fetch do not commute if they share the same target folder, nor do a Store and a Copy if they share the same target messages. However, Append and Store do commute if they have different target message UndoIDs, as do Append and Fetch.

4.2 E-mail proxy

The e-mail proxy is responsible for intercepting all SMTP and IMAP traffic directed at the mail server, converting it into the verbs described above, and interacting with the undo manager. The proxy is one of the simpler components of the undo system. It accepts connections on the IMAP and SMTP ports and dispatches threads to handle each incoming connection. Each connection is handled by a separate thread, which runs in a loop, decoding each incoming SMTP or IMAP interaction, packaging it into a verb, and invoking the undo manager to sequence, execute, and record the verb. For IMAP connections, the proxy never interacts directly with the server; it merely opens connections that are used by the verbs themselves during original or replay execution.

The SMTP case is more complicated, however. Because we want to be able to use Undo to recover from configuration errors that cause mail to be erroneously rejected, we must create verbs for all delivered messages even if they would normally be rejected. Thus the SMTP proxy acts more as a server than a proxy, completing a transaction with the client before packaging it into a verb and sending it to the real server. By doing so, however, the proxy opens itself up to denial-of-service attacks: an external attacker can generate streams of invalid SMTP requests (such as relaying requests), cluttering up the timeline log and burning extra resources to handle the later failures when those requests are executed against the real server. In our prototype, we err on the side of caution by validating recipient addresses against the real server before accepting a transaction from the client, rejecting any *syntactically*-invalid recipients or relay requests, while allowing otherwise-rejected recipients. This decision reflects a tradeoff between attack vulnerability and the system's ability to recover from configuration errors affecting address validation, and is probably a decision best left to site policy.

4.3 Time-travel storage layer

At the base of the undoable e-mail system is the time-travel storage layer, which provides stable storage for the e-mail store's hard state as well as the ability to physically restore previous versions of that state. The storage layer is designed to be application-neutral, and has neither knowledge of the e-mail store nor any customizations to e-mail semantics.

Ideally, a time-travel storage layer would provide the ability to restore state backward to any arbitrary point in time; to restore state *forward* in time to cancel the effects of a previous rollback (essential in providing the ability to undo an undo); and to accomplish these time-travel operations instantaneously. Unfortunately, we could find no storage layer offering all of these properties, so we were forced to improvise. We started out with a Network Appliance filer whose WAFL file system and SnapRestore feature provide snapshots that can be created and restored almost instantaneously. Two limitations had to be addressed: its 31-snapshot limit, and the fact that restoring an old snapshot annihilates any later ones, preventing forward time travel.

We began by building a Java wrapper that hides the telnet/console-based command-line interface to the filer's snapshot management tools. The wrapper tracks the filer's active snapshots and provides an API for creating, deleting, restoring, and listing them. Also, during normal operation, the wrapper periodically takes snapshots at multiple configurable granularities (e.g., every 10 minutes, every hour, every day, every week), aging out old ones according to an algorithm that preserves a specified minimum number of snapshots at each granularity while maximizing the number of snapshots in the most recent past. With our default granularities, the 31 available snapshots span up to a month of time, with up to 20 snapshots concentrated in the past day.

To address the lack of forward time-travel, we added a routine to the wrapper that copies an old snapshot forward to the present, effectively overwriting the current state of the system with an older snapshot of state, but without destroying any intervening snapshots. By leveraging the filer's ability to forward-restore a single file from a snapshot in constant time, this copy-forward routine runs in time proportional to the number of files in the file system, independent of their size. Given this ability, we implement reverse time-travel by first taking a recovery snapshot, then copying the desired old snapshot to the present. To do forward time-travel after that, we need merely restore the recovery snapshot, which takes the system back to the point before the old snapshot was made current.

Finally, to address the limited number of snapshots, we designed the undo manager to implement Rewind by first restoring to the nearest snapshot prior to the rewind

target, then using the existing Replay code to roll the system forward to the exact target time point. Given this approach, extra snapshots become a performance optimization rather than a functionality issue.

4.4 Undo manager

Our implementation of the undo manager is a reasonably straightforward translation of its description in Section 3 into Java code. The undo manager stores the system timeline as a linear append-only log of verbs. The log is implemented as a BerkeleyDB 'recno'-style database, with each verb assigned a sequential log sequence number (LSN). The LSN is the fundamental internal representation of time to the undo manager, and all "time-travel" operations like rewinding and replaying operate in terms of LSNs, although versions of all external interfaces are provided that take real dates and times rather than LSNs.

The undo manager mediates execution of verbs during normal operation much as described in Section 3.2. One special case bears mention: when verbs arrive for execution during an in-progress undo cycle, the execution process has to proceed differently. The undo manager cannot allow the verb's operation to modify the state of the service system, since the verb is effectively in a different timeline than the system. However, we do not want the undo system to lose delivered e-mail during an undo cycle. Similarly, we want to retain the ability for users to at least inspect their mailbox state, even if it is temporarily inconsistent and immutable (although this is again likely a site policy choice). Our solution is to defer execution of asynchronous verbs (like SMTP deliveries) until the undo cycle completes—being asynchronous, they can tolerate the delay—and to execute synchronous verbs in a read-only mode. If a synchronous verb cannot be executed read-only, the execution fails and ideally reports an explanatory message back to the user. Synchronous verbs executed read-only are still added to the end of the timeline log, as they can externalize state even if they cannot change it.

5 Analysis of Overhead and Performance

With an implementation in place, we set up some simple experiments to gauge the overhead of adding our proof-of-concept Undo implementation to an existing e-mail store and to evaluate its performance. Since Three-R's undo is targeted at reducing human operator stress and improving overall system dependability, a true evaluation of Undo would require a dependability benchmark incorporating human subjects as described in [2]; such a study is beyond the scope of this paper, although we are in the process of performing one as future work.

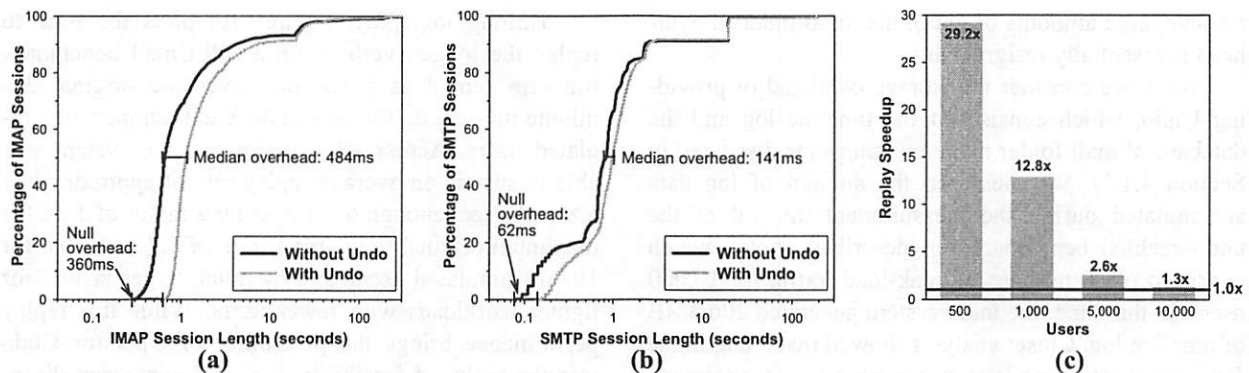


Figure 3: Overhead and Performance of Undo. The leftmost two graphs show cumulative distributions of session length for mail retrieval via IMAP (a) and mail delivery via SMTP (b), with and without the undo system in place. The rightmost graph (c) shows the performance of replay, represented as speedup over the original execution of the benchmark run.

5.1 Setup

We deployed a setup consisting of four machines: a mail store server, the undo proxy, a workload generator, and a time-travel storage server. Details of the machine configurations are given in Table 2. All machines were connected by switched gigabit Ethernet. The filer was configured with two volumes, a 250GB time-travel volume with a 40% snapshot reserve, and a 203GB log volume with the standard 15% reserve. The proxy was configured to store its timeline logs on the filer's log volume, accessed via NFS. The mail server was configured with 10,000 user accounts, with all of their storage (home directories and mailpools) placed on the filer's time-travel storage volume and accessed via NFS.

Our measurement workload was provided by a modified version of the SPECmail2001 e-mail benchmark [29]. SPECmail simulates the workload seen by a typical ISP mail server with a mixture of well-connected and dialup users. We modified the SPECmail bench-

mark to use IMAP instead of POP for retrieving mail and added code to export detailed timings for each e-mail session along with the benchmark's usual summary statistics; we also modified the benchmark to direct all mail to the mail store rather than to remote users, as we were only interested in mail store behavior. The benchmark was set up with its standard workload for 10,000 users at 100% load, a configuration that is intended to generate a workload equivalent to what a 10,000-user ISP would see during its daily load peak. In our experiments, this translated to an average of 95 SMTP connections and 102 IMAP connections per minute. Each benchmark run consisted of a 30 minute measurement interval preceded by a 3-minute warm-up.

5.2 Results: overhead

We begin by comparing the user-visible latency with and without the undo system in place. Figures 3(a) and (b) plot the cumulative distributions of the IMAP and SMTP session lengths measured both from the unmodified e-mail store server and from the undo-enabled version of the same. In the latter case, the undo system was actively proxying connections and recording the system timeline. Note that here, a session is defined as a single complete set of client interactions with the mail server, from login to logout.

Looking at the session-length distributions, we see that the Undo system does not significantly alter the shapes of the distributions, essentially just shifting them to the right. This shift represents the overhead added by the proxying, verb-generation, and verb-scheduling code. While the overhead imposed by Undo is not negligible, ranging from 62ms for a null SMTP session to 484ms for the median IMAP session, it is still relatively small compared to the threshold at which human users begin to perceive unacceptable sluggishness, typically pegged at about one second [16]. Furthermore, this latency is spread across an entire session and not a single interaction. For longer sessions (typically, those that

Machine	Configuration
Mail store server	Type: IBM Netfinity 5500 M20 CPU: 4x500MHz Pentium-III Xeon DRAM: 2GB OS: Debian 3.0 Linux, 2.4.18SMP kernel Software: Sendmail 8.12.3 SMTP server, UW-IMAP 2001.315 IMAP server
Undo proxy	Type: Dell OptiPlex GX400 CPU: 1x1.3GHz Pentium-IV DRAM: 512MB OS: Debian 3.0 Linux, 2.4.18SMP kernel Software: Sun Java2 SDK version 1.4.0_01
Workload generator	Type: IBM Intellistation E Pro CPU: 1x667MHz Pentium-III DRAM: 256MB OS: Windows 2000 SP3 Software: Sun Java2 SDK version 1.4.1
Time-travel storage server	Type: Network Appliance Filer F760 DRAM: 1GB OS: Data OnTAP 6.2.1 Disk: 14x72GB 10kRPM FC-AL, 1TB total

Table 2: Machine configurations for overhead experiments

retrieve large amounts of data), the undo-induced overhead is essentially insignificant.

Next, we consider the storage overhead of providing Undo, which consists of the timeline log and the database of mail folder name mappings (as discussed in Section 4.1.1). We measured the amount of log data accumulated during the measurement interval of the undo-enabled benchmark run described above, which consisted of 30 minutes of peak-load traffic for 10,000 users. In that time, the undo system generated 206.5MB of timeline log. Closer analysis showed that a bug in the Java serialization code was contributing an enormous amount of overhead by writing large swaths of garbage data to the log. With this overhead factored out, the undo system generates an estimated 96.2MB of uncompressed timeline log over the 30 minute interval, 71% of which consists of copies of incoming e-mail. This result extrapolates to 0.45GB of timeline log per 1,000 simulated users per day. Translating to a more concrete reference, a single 120GB disk could hold just under 250,000 user-days of log data, enough to record 3½ weeks of timeline for a 10,000-user ISP. Adding log compression may help further reduce the storage overhead of undo.

The name database size is relatively static and proportional to the number of total mail folders in the system. For our 10,000 users each of whom only had an Inbox, the corresponding name database required 12.3MB of disk space, indicating that a 120GB disk could hold the names for over 93 million e-mail folders.

5.3 Results: performance

We next look at the performance of the Three-R's cycle itself. We measured this by starting with the system at the end-state of a 30-minute SPECmail benchmark run for various numbers of simulated users, and recorded the time it took to rewind the system back to a storage checkpoint taken at the start of the benchmark run, then to replay it forward to the end of the run.

With the forward-time-travel workaround of Section 4.3 in place, it took on average 590 seconds, or 9m50s, to rewind the 10,000-user system (average of three runs, standard deviation <1%). The bulk of this time was spent copying files from the old snapshot into the active system, and so this time is heavily dependent on the number of files in the file system; our experiments show it to scale roughly linearly with the number of simulated SPECmail users. In contrast, using the Network Appliance filer's built-in snapshot restore capabilities, an old snapshot can be (non-undoably) restored in a constant 8 seconds on average (10% std. deviation over 12 runs), independent of the number of simulated users. This is the order of magnitude rewind time that would be achievable in practice, given the proper interfaces into the filer to support undoable snapshot restore.

Turning to replay, Figure 3(c) plots the time to replay the logged verbs from a SPECmail benchmark run represented as a speedup over the original 30-minute run-length, for several different numbers of simulated users. Across all experiments, the system was able to sustain an average replay rate of approximately 8.8 verbs/sec, enough to surpass by a factor of 1.3x the maximum original verb arrival rate of 6.7 verbs/sec for 10,000 simulated users, and by much larger factors for lighter workloads with fewer users. While this replay performance brings the possibility of Operator Undo into the realm of feasibility, it is still somewhat disappointing. Analysis shows that the measured replay performance is primarily due to the overhead of establishing, authenticating, and tearing down SMTP and IMAP connections for each replayed verb. Significant improvements in replay speed could be realized through more optimized connection management, and likewise if these protocols provided a "batch mode" that allowed a trusted entity (like the undo system) to reuse a single authenticated connection to replay the interleaved interactions of multiple users.

6 Related Work

Our Three-R's Undo approach draws on a host of well-studied techniques—service proxying, operation logging and replay, replica consistency management, timeline history management, and checkpoint recovery, to name a few—and creates a novel synthesis of them in the form of a tool for creating a forgiving environment for system operators. In particular, our system uniquely combines the ability to integrate repairs into a logged operation history, common in collaborative productivity application frameworks, with the system-wide applicability of traditional system checkpointing or backup/restore techniques.

It is this ability to restore history after repairs that differentiates our undo system from other log/replay-based recovery environments like transactional database systems [17] and transparent log-based rollback-recovery systems [1] [9] [15]. In these other systems, replay is only possible if the system context has not changed since the log was recorded, for they provide no mechanism to handle replayed events that fail or produce different externally-visible results than during their original execution. In transaction systems, for example, transactions are assumed to be permanent once committed, and cannot change their results or commit status as part of recovery. In log-based rollback recovery, once state escapes to the external world, rollback beyond that escape point is simply disallowed. In contrast, our Three-R's approach allows the trajectory of replay to differ from the original execution, detecting significant differences and compensating for externally-visible

inconsistencies. This detection and compensation is made possible by our verbs: not only do they provide a framework for specifying consistency and compensation, but they provide a higher-level record of user intent than transactions or message logs, making intelligent compensation more feasible.

As we have already discussed in Section 3.2.1, the challenge of retroactively integrating repairs into a logged operation history bears a great deal of similarity to the problem of reconciliation in optimistic replication systems such as Bayou, IceCube, or Coda [12] [28] [30]. But it also has an even more direct counterpart in work on timeline management for collaborative productivity applications, an area which has explored sophisticated undo models supporting highly-malleable views of time. Probably the best example of work in this area is Edwards and Mynatt's Timewarp system [8], a framework for collaborative productivity applications that maintains histories of all user actions over shared state. In Timewarp, users can rewind their state, alter their histories, and replay changes at will. Timewarp defines a framework for detecting and managing inconsistencies that arise from retroactively-inserted changes to the timeline, much as we do to handle inconsistencies resulting from Repair, but Timewarp's approach requires that all alterations to the past timeline consist of insertions or deletions of predefined actions in the operation log, as it identifies inconsistencies by detecting conflicts between the well-known actions [7]. In contrast, our design point of allowing unconstrained repair limits the applicability of the Timewarp approach, and hence we detect inconsistency by directly examining externalized state. Along similar lines, Timewarp performs undo (Rewind) logically, whereas we must perform it physically as we cannot trust that operations were processed correctly during original execution.

Our Three-R's Undo approach supports recovery from system-wide problems, not just errors within an application. Again, this property on its own is available in many other systems. Users of desktop PCs can purchase software tools such as Roxio's GoBack [26] or IBM/XPoint's Rapid Restore [32] that provide the ability to examine past system states, physically roll-back an entire machine, including the OS, to a past state, and even roll-forward again later. Virtual machine systems such as VMware [31] provide the ability to log system operation so it can be rolled-back and replayed; Dunlap et al.'s ReVirt system demonstrates a particularly clever use of the technique for intrusion analysis [6]. But unlike our Operator Undo model, these systems provide either Repair or Replay, but never both—if changes or repairs are made to a rolled-back system, replay either wipes out those changes or is prohibited altogether.

In terms of our actual implementation of an undoable e-mail system, probably the most relevant prior work is a commercial product, the Network Appliance SnapManager for Exchange, which is a system that integrates the snapshot capabilities of Network Appliance filers with Exchange's built-in mail logging [19]; similar functionality is provided by other systems that build e-mail atop a database system. In the case of a system failure, the SnapManager system allows an operator to restore a previously-archived snapshot of the mail system, then replay forward using the Exchange transaction logs. While this system is similar in approach to our Three-R's-undoable e-mail store, there are two fundamental differences. First, the SnapManager system does not detect and compensate for external inconsistencies. Second, operation logs in the SnapManager/Exchange system are recorded deep within the Exchange system, long after the user's protocol interactions have been processed. If the Exchange server is misconfigured or buggy, these logs may be incomplete or corrupted, and will almost certainly not contain a record of mail deliveries incorrectly rejected by the system. In contrast, our external proxy-based logging takes place before the e-mail server even interprets the mail protocols, allowing recovery from failures or configuration problems at all levels of the mail server, and not binding the logs to a specific server implementation (hence allowing server upgrades as part of Repair). While our approach is still susceptible to bugs in the proxy, the likelihood of those bugs is reduced by the relative simplicity of the proxy and the fact that the proxy executes operations directly from the same verb objects as are stored in the timeline log, quickly flushing out bugs during normal operation.

Finally, while we have focused entirely on the mail store and its operator, there has been work on developing undo-like functionality for the user side of e-mail. Here, the challenge is not recovery from system operator error or software bugs, but dealing with user errors like accidental mailbox deletion. We think that many of our Three-R's undo techniques would make sense at this level of the system, and are thinking about ways to extend our approach to provide undo to both operators and end-users, but unlike some brave researchers [27], we do not intend to tackle the most challenging of end-user problems: unsending e-mail on the Internet.

7 Conclusions and Future Work

Dependable systems will not be achieved until we address the challenges facing the human operators who exert such a crucial influence on dependability. Our model of an operator-targeted system-wide undo facility is a first step toward creating a forgiving environment for system operators so that they may better respond to system problems. It reduces the impact of mistakes,

allowing for inevitable human error and making trial-and-error solutions feasible, and provides a last-resort tool for guaranteed recovery from developing software problems or data corruption.

Our proof-of-concept implementation of undo for e-mail proves the concept feasible and demonstrates that even an unoptimized implementation imposes reasonable overheads in terms of space and time. That said, there are several future directions we are pursuing to increase the functionality of Operator Undo and to better understand its influence on dependability. First, we are considering extensions to the basic undo model that would allow for more complex undo timelines supporting multiple branches of history. We are also working to extend the model to support multiple hierarchical levels of undo, allowing undo to be simultaneously exposed at per-user, per-machine, and per-cluster granularities. We would like to see implementations of Three-R's-style undo in more applications than just e-mail; we are considering the design of an undoable auction service, and would welcome company in exploring other applications. Finally, we feel it is important to understand how a tool like Undo affects the behavior of system operators and how those behavioral changes impact dependability; to that end, we are developing dependability benchmarks that incorporate human operators as participants.

Source code

Source code for our undo framework and e-mail proxy is available at <http://roc.cs.berkeley.edu/undo/>.

Acknowledgements

This work was supported in part by DARPA under contract DABT63-96-C-0056, the NSF under grant CCR-0085899 and infrastructure grant EIA-9802069, and the California State MICRO Program. The first author was supported by an IBM Graduate Fellowship. Special thanks go to Network Appliance for their generous donation of the filer used in our experiments, to Mike Howard for always having an experimental testbed or two up his sleeve, and to our shepherd, Brian Noble, and our anonymous reviewers for their feedback and insight.

References

- [1] A. Borg, W. Blau et al. Fault Tolerance Under UNIX. *ACM TOCS*, 7(1):1–24, February 1989.
- [2] A. Brown, L. C. Chung, and D. A. Patterson. Including the Human Factor in Dependability Benchmarks. *Proc. 2002 DSN Workshop on Dependability Benchmarking*. Washington, D.C., June 2002.
- [3] A. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to Dependability. *Proc. 10th ACM SIGOPS European Workshop*. St. Emilion, France, 2002.
- [4] A. Brown and D. A. Patterson. To Err is Human. *Proc. 2001 Workshop on Evaluating and Architecting System Dependability*. Göteborg, Sweden, July 2001.
- [5] M. Crispin. *RFC2060: Internet Message Access Protocol Version 4rev1*. December 1996.
- [6] G. Dunlap, S. King, et al. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *Proc. 5th OSDI*. Boston, MA, December 2002.
- [7] W. K. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. *Proc. 10th ACM Symp. on User Interface Software and Technology*. Banff, Canada, October 1997.
- [8] W. K. Edwards and E. D. Mynatt. Timewarp: Techniques for Autonomous Collaboration. *ACM Conf. on Human Factors in Computing Systems*. Atlanta, GA, 1997.
- [9] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *CMU TR 96-181*. Carnegie Mellon, 1996.
- [10] J. Gray. Why Do Computers Stop and What Can Be Done About It? *Symp. on Reliability in Distributed Software and Database Systems*, 3–12, 1986.
- [11] IBM. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
- [12] A. Kermarrec, A. Rowstron, et al. The IceCube approach to the reconciliation of divergent replicas. *Proc. 20th ACM Symp. on Principles of Distributed Computing (PODC 2001)*. Newport, RI, August 2001.
- [13] J. Klensin, ed. *RFC2821: Simple Mail Transfer Protocol*. April 2001.
- [14] R. Lemos and M. Farmer. Microsoft fingers technicians for crippling site outages. *ZDNet News*, 25 January 2001.
- [15] D. E. Lowell, S. Chandra, and P. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. *Proc. 4th OSDI*. San Diego, CA, October 2000.
- [16] R. B. Miller. Response time in man-computer conversational transactions. *Proc. AFIPS Fall Joint Computer Conference*, 33:267–277, 1968.
- [17] C. Mohan, D. Haderle, et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Systems*, 17(1): 94–162, 1992.
- [18] B. Murphy and T. Gent. Measuring System and Software Reliability using an Automated Data Collection Process. *Quality and Reliability Engineering International*, 11:341–353, 1995.
- [19] Network Appliance. SnapManager Software. <http://www.netapp.com/products/filer/snapmanager2000.html>.
- [20] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? *Proc. 4th USENIX Symp. on Internet Technologies and Systems*. March, 2003.
- [21] Osterman Research. Survey on Messaging System Downtime from a user perspective. http://www.ostermanresearch.com/results/surveyresults_dt0801.htm.
- [22] D. A. Patterson, A. Brown, et al. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. *UC Berkeley TR UCB/CSD-02-1175*. Berkeley, CA, March 2002.
- [23] N. Preguiça, M. Shapiro, and C. Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. *Microsoft TR MSR-TR-2002-52*. May 2002.
- [24] J. Reason. *Human Error*. Cambridge University Press, 1990.
- [25] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. *Proc. 18th SOSP*. Banff, Alberta, Canada, October 2001.
- [26] Roxio, Inc. GoBack3. <http://www.roxio.com/en/products/goback/index.jhtml>.
- [27] A. Rubin, D. Boneh, and K. Fu. Revocation of Unread E-mail in an Untrusted Network. *Proc. 1997 Australasian Conf. on Information Security and Privacy*. Sydney, Australia, July 1997.
- [28] M. Satyanarayanan. The Evolution of Coda. *ACM Transactions on Computer Systems*, 20(2):85–124, May 2002.
- [29] SPEC. SPECmail2001. <http://www.spec.org/osg/mail2001>.
- [30] D. B. Terry, M. M. Theimer, et al. Managing Update Conflicts in a Bayou, a Weakly Connected Replicated Storage System. *Proc. 15th SOSP*. Copper Mountain, CO, Dec. 1995.
- [31] VMware. <http://www.vmware.com>.
- [32] Xpoint Technologies. XPoint Rapid Restore Server. <http://www.xpointdirect.com/en/IBMRRPC/RRServer.asp>.

Role Classification of Hosts within Enterprise Networks Based on Connection Patterns

Godfrey Tan
MIT
godfreyt@mit.edu

Massimiliano Poletto
Mazu Networks
maxp@mazunetworks.com

John Guttag and Frans Kaashoek
MIT
{guttag,kaashoek}@mit.edu

Abstract

Role classification involves grouping hosts into related roles. It exposes the logical structure of a network, simplifies network management tasks such as policy checking and network segmentation, and can be used to improve the accuracy of network monitoring and analysis algorithms such as intrusion detection.

This paper defines the role classification problem and introduces two practical algorithms that group hosts based on observed connection patterns while dealing with changes in these patterns over time. The algorithms have been implemented in a commercial network monitoring and analysis product for enterprise networks. Results from grouping two enterprise networks show that the number of groups identified by our algorithms can be two orders of magnitude smaller than the number of hosts and that the way our algorithms group hosts highly reflect the logical structure of the networks.

1 Introduction

Today, many enterprises have internal networks (intranets) that are as or more complicated than the entire Internet of a few years ago. Managing these networks is increasingly costly, and the business cost of network problems increasingly high.

Managing an enterprise network involves a number of inter-related activities, including:

Establishing a topology. A network's topology has a significant impact on its cost, security, and performance. An increasingly important aspect of topology design is *network segmentation*. In an effort to provide fault isolation and mitigate the spread of worms like Nimda [3] and Code Red [2], enterprises segment their networks using firewalls [4], routers, VLANs [7], and other technologies.

Establishing policies. Different users of a network have different privileges. Some users may have

unlimited access to external networks while others may have restricted access. Some users may be limited in the amount of bandwidth they may consume, and so on. The number of policies is open-ended.

Monitoring network performance. Almost every complex network suffers from various localized performance problems. Network managers must detect these problems and take action to correct them.

Detecting and responding to security violations.

Increasingly, networks are coming under attack. Sometimes the targets are chosen at random, as in most virus-based attacks, and in other cases they are picked intentionally, as with most denial-of-service attacks. These attacks often involve compromised computers within the enterprise network. Early detection of attacks plays a critical role in reducing the damage.

Conducting these activities on a host-by-host basis is not feasible for large networks. Network managers need to extract structure from their networks so that they can think about them and make decisions at larger levels of granularity. Today, this structuring is most often done in an *ad hoc* manner that relies on administrators' best guesses about the computers, services, and users on the network. Obviously, this method has scaling problems.

This paper presents two algorithms that, used together, partition the hosts on an enterprise network into groups in a way that exposes the logical structure of a network. The *grouping algorithm* classifies hosts into groups, or "roles," based on their *connection habits*. The *correlation algorithm* correlates groups produced by different runs of the classification algorithm.

The two algorithms together provide the following properties:

1. They guarantee that a host is only grouped with other hosts that have the strongest degree of similarity in connection habits.

2. They provide a mechanism to merge groups, and give network administrators fine-grained control over the merging process, so that meaningful results can be achieved.
3. They deal with transient changes in connection patterns by analyzing the profiled data over long periods.
4. They respond to non-transient changes in connection patterns by producing a new partitioning and describing the differences between the new partitioning and the previous partitioning.
5. Their run time grows quadratically with the number of hosts in the enterprise network.

As we demonstrate in Section 6, the algorithms reduce the number of logical units that a network administrator must deal with by *one or two orders of magnitude*. The algorithms are implemented as part of an enterprise monitoring and analysis system that is in production use at several large enterprises.

Section 2 outlines the system in which the algorithms operate, and introduces an example scenario that will be used throughout the paper. Section 3 describes the models used to develop practical solutions. Section 4 and Section 5 explain the two practical algorithms for solving the role classification problem. Section 6 presents preliminary results, and Section 7 discusses related work. We conclude with discussions of our current and future work in Section 8.

2 System Overview

The role classification algorithms are implemented as part of a system designed to detect and respond to security violations in large enterprise networks. Such networks commonly consist of tens of thousands of computers, spread over different geographic locations. The security system consists of *probes* and a central *aggregator*. The probes analyze packets on the link or links they are attached to, and send relevant information (including IP address/port tuples) to the aggregator.

The aggregator is a scalable system that consists of one or more CPUs. It periodically runs several analysis algorithms on the data it has received from the probes. It uses the role classification algorithms to refine its analyses and to allow the administrators to describe group-based policies.

Figure 1 presents a simple enterprise network and a partitioning of computers into groups that the aggregator might produce based on the communication patterns observed by the probes. The communication patterns

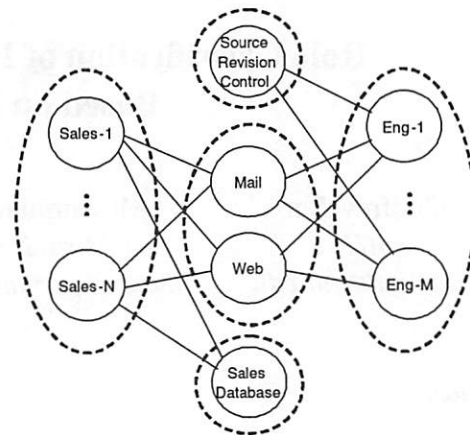


Figure 1. Grouping of related hosts based on connection patterns. Edge indicates that nodes communicate regularly. The dashed circle represents the group boundary.

might indicate that hosts *Sales-1* to *Sales-N* communicate with three servers: *Mail* server, *Web* server, and *SalesDatabase* server. Similarly, the patterns might indicate that hosts *Eng-1* to *Eng-M* communicate mostly with *Mail* server, *Web* server, and *SourceRevisionControl* server.

Based on this information the grouping algorithm can logically divide all machines into five groups: (i) the sales group consisting of hosts *Sales-1* to *Sales-N*, (ii) the engineering group consisting of hosts *Eng-1* to *Eng-M*, (iii) the common server group consisting of *Mail* and *Web*, (iv) the sales server group consisting of *SalesDatabase* and (v) the engineering server group consisting of *SourceRevisionControl*.

The results of the grouping algorithm are currently being used in two major ways:

1. The Mazu network monitoring and detection system decides whether a host's behavior matches the expected policy setting, partly based on the history of the host's group membership. For example, if a host in the engineering group were to suddenly start opening connections to the *SalesDatabase* server, it might be a cause for alarm.
2. The network administrators review the grouping results to better understand the structure of their networks and to get useful insights for conducting network re-organization tasks such as consolidating servers and network segmentation.

The system allows a network manager to label each identified group with descriptive roles and set policies

per group. The system continuously monitors the communication patterns, adjusts groups as computers come and go, flags policy violations, and raises alerts about potential security violations. Because all this information is presented on the level of groups (instead of individual hosts), a network manager is able to understand and process the changes and alerts more easily. The algorithms also provide network administrators with flexibility to control the grouping process to achieve results that highly reflect their intuitive notion of the network structure.

The algorithms presented in this paper are solely based on the connection patterns of hosts such as the set of neighboring hosts. However, the algorithms can easily be extended to use other information such as protocols and port numbers used and bytes transferred to achieve desired results. For instance, some network administrators may desire that *Mail* and *Web* servers be put in different groups. In this case, the protocol information can be used to keep the role classification algorithm from grouping together hosts that use different sets of protocols. We are currently exploring ways to expand the capability of the grouping and correlation algorithms by providing network administrators with more flexibility to achieve desired results.

The algorithms assume that the connection patterns of hosts highly reflect the logical roles that they play. For some networks where this is not true, the algorithms will not do a good job. However, we believe that hosts in a typical enterprise network that share the same logical role will demonstrate similar connection patterns.

3 Model

In this section, we develop a model for thinking about the grouping problem. We define the problem in the abstract, providing a model with several functions and parameters that can be adjusted to meet various goals. Later in the paper, we present and evaluate instantiations of these parameters.

- Let I be the set of hosts in an enterprise network. We will use $|I|$ to denote the number of hosts in I .
- Let *similarity* be a commutative function from pairs of hosts in I to an integer greater than or equal to 0. Roughly speaking, if *similarity*(h_1, h_2) is high, then we would like our grouping algorithm to place the hosts h_1 and h_2 in the same *group*. Defining *similarity* so that it is both efficient to compute and yields a good grouping is at the heart of the problem addressed in this paper.
- A partitioning P of I *respects similarity* if for all distinct groups $G_1, G_2 \in P$, $h_1, h_2 \in G_1$, and

$$h_3 \in G_2,$$

- $\text{similarity}(h_1, h_2) \geq \text{similarity}(h_1, h_3)$
- $\text{similarity}(h_1, h_2) \geq \text{similarity}(h_2, h_3)$

We extend this definition of *similarity* to define the average similarity between a host h_1 and a group G_2 , $\text{avg_similarity}(h_1, G_2)$, as the ratio of the sum of the similarity between h_1 and each $h_2 \in G_2$ to the number of hosts in G_2 :

$$\text{avg_similarity}(h_1, G_2) = \frac{\sum_{h_2 \in G_2} \text{similarity}(h_1, h_2)}{|G_2|}$$

A partitioning P of I *respects avg_similarity* if for all $h_1 \in G_1$ and $G_2 \in P$, $\text{avg_similarity}(h_1, G_1) \geq \text{avg_similarity}(h_1, G_2)$.

Respecting *similarity* or *avg_similarity* is not sufficient to generate a useful partitioning of I . After all, a partitioning that puts all the nodes in one group or one that puts each node in a separate group respects *similarity*. We therefore provide a parameter that can be used by network administrators to control how aggressive the algorithm is in partitioning I into groups.

- Let S_{\min} , the *similarity threshold*, be an integer greater 0. A partitioning respects *similarity* and S_{\min} if it respects *similarity* and if, for h_1 and h_2 in G , $\text{similarity}(h_1, h_2) \geq S_{\min}$.
- A partitioning P of I is said to be *maximal* with respect to *similarity* and S_{\min} if it respects *similarity* and S_{\min} and there does not exist another partitioning of I that respects *similarity* and S_{\min} and has fewer groups. By adjusting S_{\min} , one gets a maximal grouping with fewer groups in which the members of each group are more similar to each other.

3.1 Defining Similarity

We use connection behavior as a basis for host grouping, because that information is easily available by just monitoring the network. To group hosts, we need to define similarity in a way that captures the extent to which pairs of hosts establish connections to the same set of other hosts. We start by defining similarity between hosts as a function of the number of common hosts with which they communicate. Intuitively, hosts that share the same logical role communicate with similar sets of hosts.

A *connection* is a pair consisting of a source host address and a destination host address. The connection set of a host, $C(h)$, is the set, $\{a \mid a \in I \text{ and there is a connection between } h \text{ and } a\}$. If $h_1 \in C(h_2)$, then

$h_2 \in C(h_1)$. We define the relation $neighbor(h_1, h_2)$ to be true if and only if $h_1 = h_2$ or $h_1 \in C(h_2)$. For later use, we extend the definition of $neighbor$ to groups by defining $neighbor(G_1, G_2)$ to be true if and only if there exists a host $h_1 \in G_1$ that is a neighbor of another host $h_2 \in G_2$.

We can use the notion of a connection set to provide a simple definition of *similarity*:

$$similarity(h_1, h_2) = |C(h_1) \cap C(h_2)| \quad (1)$$

That is to say that $similarity(h_1, h_2)$ is equal to the number of neighbors that h_1 and h_2 have in common.

We are now in a position to specify the requirements of a grouping algorithm. Given a set of hosts, I and a similarity threshold, S_{min} , it must find a partitioning, P , of I that is maximal with respect to *avg_similarity* and S_{min} , i.e.,

1. P respects *avg_similarity*. This constraint guarantees that each host is within the group with which it has the strongest average similarity.
2. For all $h \in G$ and $G \in P$, $avg_similarity(h, G) \geq S_{min}$. This requirement guarantees that each host in a group is sufficiently closely related to every other host in the group, thus ensuring that groups are not too large.
3. There is no other partitioning P of I that meets the first two requirements and has a larger average group size. This guarantees that groups are not too small.

This specification is independent of the definition of *avg_similarity*. For some networks, such as the one represented in Figure 1, the above definition of *avg_similarity* yields excellent results. However, for others a slightly more complex definition works better. We present such a definition in Section 4.2.

4 Role Classification

The role classification problem is not difficult to solve in ideal situations, such as the network shown in Figure 1, in which two nodes that share the same logical role communicate with the identical set of machines. Clearly, such a situation does not reflect the connection patterns in typical enterprise networks. Three major challenges of the role classification problem are:

1. Two hosts that share the same logical role may communicate with drastically different sets of machines.
2. A host may potentially be classified into more than one role.

3. The grouping results that network administrators desire may vary from network to network and therefore the role classification algorithm must provide flexibility for them to control its mechanics so that meaningful grouping results can be achieved.

In a typical network setting for a technology company, each lab or test machine may be dedicated to a single engineer. Thus, each of these lab machines, despite sharing the same role, can have a connection pattern that is very different from the rest of the lab machines. To be able to correctly group such machines together, the grouping algorithm must take into account the potential roles of neighboring hosts rather than comparing the neighbor sets.

Furthermore, some hosts may potentially be classified into more than one role. For instance, there could exist a machine in the network in Figure 1 that communicates with both sets of machines with which many engineering machines and sales machines communicate respectively. In such cases, the connection patterns of hosts must be evaluated carefully to ensure that each host is grouped with other hosts with which it has the strongest similarity in connection habits.

The role classification problem is not trivial for the aforementioned reasons. Not only does the computation of the similarity measure matter, but the process of how nodes are grouped based on the similarity values among node pairs is also important.

The grouping algorithm consists of two phases: i) the group formation phase and ii) the group merging phase. The group formation phase identifies each group of hosts that have similar sets of neighbors using a simple similarity measure such as the one described in Section 3. The purpose of the group formation phase is two-fold: i) to efficiently identify various groups of hosts, each of which has drastically different overall connection patterns, and ii) to prepare for the second phase of the algorithm. The formation phase of the algorithm can efficiently find the desired partitioning for the example network in Figure 1 but may fail for many networks since it does not take into account the potential roles of neighboring hosts as explained earlier. In general, the group formation phase may generate a partitioning that contains more groups than desired.

The group merging phase decides whether groups, produced by the formation phase, can further be merged using a much more sophisticated similarity measure. This phase provides network administrators with fine-grained control over the merging process so that the grouping results reflect their intuition of the network structure.

4.1 Forming Groups

Group formation can be thought of as a graph theory problem. From the connection sets of I , one can generate a *neighborhood* graph, *nbh-graph*, where each node represents a host and each edge with weight e represents that there are e common neighbors between the hosts. Thus, a neighborhood graph captures the extent to which pairs of hosts communicate with the same set of other hosts. We use an undirected graph since almost all communication between hosts in the intranets is bi-directional. However, in certain situations, directionality may be used to improve the quality of the grouping results and we continue to investigate this issue.

One approach to the grouping problem is to treat it as a k -clique problem where *nbh-graph* is partitioned into cliques of size k in which each edge in the clique has a weight greater than or equal to some constant c . Once a k -clique is identified, one assigns all the nodes in the k -clique to one group, since they all share at least c common neighbors. This approach is problematic, because (i) the k -clique problem is NP-complete [25], and (ii) requiring that each host pair in a group has exactly k common neighbors is too strong a requirement.

Another approach is to treat grouping as related to the problem of identifying bi-connected components (BCCs). A BCC is a connected component in which any two edges lie in a simple cycle. Thus, there exist at least two disjoint paths between any two nodes in a BCC. Unlike the k -clique problem, BCC can be solved in $O(N + E)$, where N and E are the number of nodes and edges in the graph respectively [9, 27]. Moreover, all nodes in the BCC need not be connected to each other directly. This approach is the one we use.

The group formation phase operates as follows:

1. Generate the connectivity graph, *conn-graph*, based on the observed connection patterns.
2. For $k = k_{max}$ down to 1, where k_{max} is the maximum number of hosts with which a single host communicates:

Repeat until no new groups can be assigned:

- (a) From *conn-graph*, build the k -neighborhood graph *k-nbh-graph*.
- (b) Remove group nodes (see 2d) from *k-nbh-graph*.
- (c) Generate all BCCs in *k-nbh-graph*.
- (d) For each BCC B , replace in *conn-graph* the nodes in B by a new group node G representing those nodes. Label G by a pair (ID_G, K_G) , where ID_G is a unique identifier

and K_G is k . (K_G will be used later to compute the degree of similarity between groups.)

- (e) For each ungrouped host h , where $k < \alpha \times |C(h)|$ and $0 \leq \alpha \leq 1$, create a new group G containing only h as described in 2d.

The algorithm runs iteratively over *conn-graph* until no ungrouped node remains or $k = 0$. At each step multiple BCCs may be identified simultaneously and a single node could be a part of several BCCs indicating that it may share multiple roles. In this case, the node becomes a part of a BCC with the largest size. If more than one such BCC exists, we choose one randomly. By iterating over k from high to low, the algorithm associates each host h with other hosts with the strongest similarity.

In the grouping algorithm, the minimum number of nodes required to form a BCC is two. Technically, the minimum number of nodes to form a BCC is 3, since we do not allow duplicate edges between any two nodes. Nevertheless, we allow two isolated nodes connected by an edge to form a group.

Since a BCC is not a clique, some node pairs in the BCC may not have edges between them allowing node pairs that do not share at least k common neighbors to be in the same group. However, any two nodes in a BCC have at least two disjoint paths along which two successive nodes share at least k common neighbors. In other words, any two nodes in a group demonstrate in at least two different ways that they have strong similarity in connection habits, significantly reducing the possibility that they may serve different roles. This observation is a major reason why we believe BCCs are suitable for forming groups.

When a set of hosts is placed into a group, the nodes representing those hosts are removed from *conn-graph* and replaced by one node (called the *group node*) representing the entire group. There are edges connecting that group node to each node to which one of the hosts in the group was connected.

In some cases where a node may have connection patterns so different from any other nodes, the node should form a group by itself. Step 2e forms a new group with only h in it if there exist no nodes that have the number of common neighbors greater than or equal to $\alpha \times C(h)$. We set $\alpha = 0.6$ and find that it works well with various networks.

Figure 2 illustrates the evolution of the grouping algorithm, in terms of *k-nbh-graph*, for the network depicted in Figure 1. The first group is formed when $k = M + N$, where M is the number of hosts used by sales personnels and N is the number of hosts used by engineers. For specificity, let us assume that $M = N = 3$. As shown in the picture, the *6-nbh-graph* con-

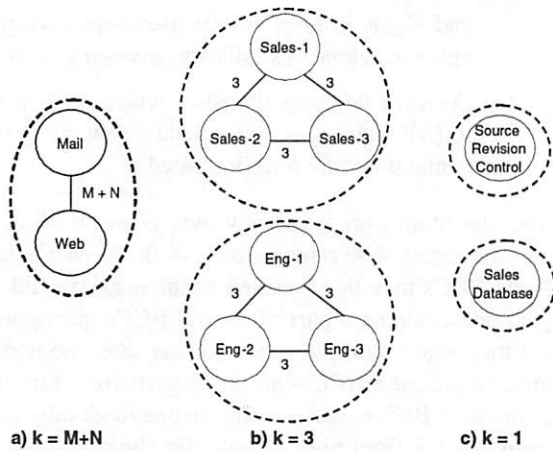


Figure 2. Evolution of the grouping algorithm at various k values.

tains two hosts, *Mail* and *Web*, and the algorithm puts them in one group. When $k = 3$, the algorithm identifies two additional BCCs, one containing all the sales machines and the other, all the engineering machines. Finally, because of the bootstrap condition (see Step 2e), the algorithm creates two groups, one containing *Sales-Database* and the other, *SourceRevisionControl*, when $k = 1 < 0.6 \times M$.

4.2 Merging Groups

The aforementioned group formation algorithm that uses a simple definition of *similarity* tends to produce too many groups in many situations. Consider, for example, the network in Figure 1 modified so that *Sales-1* communicates with the *Mail* and *SalesDatabase* servers but not the *Web* server. The grouping algorithm in Section 4 will create two groups for the sales hosts, one that only contains *Sales-1* and one that contains the rest of the sales hosts. This might be appropriate, but it is probably not what a network administrator would want.

The group merging phase builds on the results generated by the group formation phase. It merges groups that are similar in connection habits in a way that allows users to control the process so that more meaningful results can be achieved.

During the grouping phase, we merge two groups if they meet the following two requirements:

Similarity requirement. The similarity measure between the two groups exceeds a user-specified threshold.

Connection requirement. The average number of connections of each group is comparable.

```

PROCEDURE MEETCONNECTIONREQ( $G_1, G_2$ ) {
   $a1 \leftarrow \frac{\sum_{h1 \in G_1} C(h1)}{|G_1|}$ 
   $a2 \leftarrow \frac{\sum_{h2 \in G_2} C(h2)}{|G_2|}$ 
  return ( $a1$  is within  $\beta$  percent of  $a2$ )
}

PROCEDURE MEETSIMILARITYREQ( $G_1, G_2$ ) {
   $kmax \leftarrow \max(K_{G_1}, K_{G_2})$ 
   $s \leftarrow \text{SIMILARITY}(G_1, G_2)$ 
  if ( $kmax \geq K^{hi}$  and  $s \geq S_g^{hi}$ )
    return true;
  else
    return ( $kmax < K^{hi}$  and  $s \geq S_g^{lo}$ )
}

PROCEDURE SIMILARITY( $G_1, G_2$ ) {
   $c1 \leftarrow \frac{\sum_{h \in C(G_1)} CP(h, G_1)}{|G_1|}$ 
   $c2 \leftarrow \frac{\sum_{h \in C(G_2)} CP(h, G_2)}{|G_2|}$ 
  For each common neighbor group  $G'$  of  $G_1$  and  $G_2$ 
     $s \leftarrow s + \min(\frac{CP(G', G_1)}{|C(G_1)|}, \frac{CP(G', G_2)}{|C(G_2)|})$ 
   $s \leftarrow \frac{1}{2} * (\frac{s}{c1} + \frac{s}{c2})$ 
  return  $s * 100$ 
}

```

Figure 3. Pseudo-code for determining the similarity and connection requirements.

The algorithm repeatedly merges two groups that meet the two requirements and have the highest similarity measure until no groups can be merged. The K value of a newly merged group is set to the minimum number of connections a host in the group has.

Figure 3 depicts the pseudo-code for determining the average connection requirement and the similarity requirement. The procedure MEETCONNECTIONREQ decides whether the two groups, G_1 and G_2 , meet the connection requirement. This requirement keeps a group with a large number of connections from merging with another group with a much smaller number of connections.

The procedure MEETSIMILARITYREQ determines whether the two groups meet the similarity requirement. S^{hi} and S^{lo} , $S^{hi} > S^{lo}$, are similarity thresholds that can be set by the user to control the merging process. Which threshold is used depends upon whether $\max(K_{G_1}, K_{G_2}) \geq K^{hi}$, where K^{hi} is a constant intended to define whether a K_G value is "high." The similarity threshold for merging groups is higher for groups with a high K_G value, those groups whose member hosts share a high number of common neighbors. This is because merging two groups can change the relationships between other groups in a way that induces additional undesirable merges.

Again, consider the groups in the example network

illustrated in Figure 1. Notice that if N is large, the similarity measure between the *SalesDatabase* group and the *Mail* and *Web* group will be large. Similarly, for large M , the *SourceRevisionControl* group will be highly similar to the *Mail* and *Web* group. If all three groups were to merge, it would effectively cause the sales group and the engineering group to merge, resulting in a partitioning with two groups: one containing all the servers, and one containing all other hosts. In most situations, this grouping would be undesirable since the network administrators lose the important separation between the *Sales* machines and the *Eng* machines. For these reasons, groups with high K_G values are required to have a higher similarity measure to merge. We discuss how best to choose the constants in Section 6.

SIMILARITY computes the similarity s (between 0 and 100) of connection patterns between two groups. $CP(x_1, x_2)$ returns the total number of connections between x_1 and x_2 , where x_i could either be a host or a group. Two groups are considered similar if they have many common neighboring groups and similar average numbers of connections. For example, if the set of neighbors of G_1 is a subset of the set of neighbors of G_2 , it increases the desirability of merging these two groups. However, if the average numbers of connections of G_1 and G_2 are quite different, the desirability of merging them is lessened.

5 Role Correlation

Over time, connection habits may evolve as new servers and employees are added while some existing ones leave. Sometimes hosts may behave erratically as a result of being victims or villains of denial of service (DOS) attacks. Due to any of these behaviors, the grouping algorithm may produce a different set of groups than the one produced by the algorithm a few days ago. As explained in Section 4, the grouping algorithm assigns an integer ID to each group of hosts that it identifies. There is no guarantee that the sets of IDs produced by two runs of the grouping algorithm will have any correlation between them. This situation is clearly undesirable to the users who may want to associate logical names and policy settings to the group IDs and preserve these group specific data throughout the executions of the grouping algorithm at various times.

In this section, we describe in detail the group correlation algorithm that takes the two sets of results produced by the grouping algorithm and correlates the IDs of one set with that of the other so that the two groups, one in each set of resulting groups, will have the same ID if and only if the machines in both groups are highly likely to share the same logical role.

5.1 Challenges

For the rest of this section, we assume that there exists a unique host identifier that never changes. We note that the IP address may not be a good use when Dynamic Host Control Protocol (DHCP) is used since a host's IP address may change over time. For smaller networks, a simple solution such as using DNS names as unique identifiers and dynamically updating the changes of IP addresses may be sufficient [26]. This problem of assigning a unique identifier to each host within enterprise networks is beyond the scope of this paper.

The connection habits of a host may change as a result of the following events: i) new host arrivals, ii) existing host removals, and iii) role changes by existing hosts. Due to a combination of these events, some existing hosts may communicate with different sets of hosts and thus the results of the grouping algorithm before and after these events may be different as: i) new groups are formed, ii) existing groups are deleted, iii) the member compositions of some groups change, and iv) the connection sets of some groups change. The changes affect not only the hosts directly involved in the aforementioned events but also to other hosts whose connection habits have not changed in a logical sense.

Hypothetically, if we know the exact sequence of every single change event that happened between two executions of the role classification algorithm, the results of the first execution could be incrementally updated to achieve the new results. Having such a change log, although not impossible, can complicate the network data gathering process. More importantly, a detailed change log cannot always lead to correct ID correlations.

Consider the example network in Figure 1. Assume that *Sales-1* and *Eng-1* switch roles as a result of personnels switching jobs or changing machines. *Sales-1* now communicates with *SourceRevisionControl* whereas *Eng-1* communicates with *SalesDatabase*. From the change log, it would seem that the connection sets of both *SourceRevisionControl* and *SalesDatabase* change whereas in reality, their logical roles never changed. The difficulty here is in distinguishing which changes in connection patterns are the primary causes that result in differences in group formations between two executions of the grouping algorithm. Furthermore, there may also be natural changes in connection patterns of many nodes. For instance, an existing server machine may be replaced by two new machines that do load sharing among client machines. The logical roles of the client machines have not changed but their observed connection patterns have. The rest of this section describes the role correlation algorithm that does not rely on the change log but rather uses the same set of information (i.e. only connection sets) made available to

the grouping algorithm.

5.2 The Role Correlation Algorithm

The correlation algorithm operates by comparing the results of two executions of the grouping algorithms. Let P_{t-1} and P_t be the group sets generated by the grouping algorithm at time $t-1$ and t respectively. The correlation algorithm updates the ID set of P_t , so that $ID_{G_t} = ID_{G_{t-1}}$, where $G_t \in P_t$ and $G_{t-1} \in P_{t-1}$, if and only if G_t and G_{t-1} considered to represent the same logical role. More specifically, the connection patterns of the members of G_t and those of G_{t-1} are very similar. The groups correlation algorithm correlates the ID_{G_t} and $ID_{G_{t-1}}$ in a meaningful manner and thus allow applications to preserve data specific to a particular group.

The role correlation algorithm:

- Isolates the primary events, such as node arrivals and removals, that directly affects the connection habits of groups,
- Identifies nodes that have not changed their neighbors,
- Heuristically computes the *time-varying* similarity between the connection habits of two groups formed at times t and $t-1$, and assigns $ID_{G_t} = ID_{G_{t-1}}$ if and only if the role of hosts (in terms on their connection patterns) in G_{t-1} can be considered identical to the role of hosts in G_t .

First, the correlation algorithm eliminates the differences between the two host sets, I_t and I_{t-1} , so that it can compare the connection patterns meaningfully. The algorithm computes the set of nodes that existed at time $t-1$ but have been removed in time t ($I_{t-1} \setminus I_t$), and the set of nodes that only appear at time t ($I_t \setminus I_{t-1}$). All new nodes are removed from I_t and deleted nodes are removed from I_{t-1} . Thus, the changes in connection set of each host is only as a direct result of changing connection patterns between the host and its neighbors (which existed at time t).

Second, the algorithm heuristically identifies the set, H_{same} , of nodes that are very like to play the same logical roles from $t-1$ to t . We say that the two nodes h_t and h_{t-1} are highly likely to be the same machine (i.e. it hasn't changed its logical role) if they have the identical connection sets. Specifically, $H_{same} = \{h_t | \exists h_{t-1} \in I_{t-1}, C(h_t) = C(h_{t-1})\}$. We will explain shortly how we use the fact that a host $h \in H_{same}$ to our advantage in computing the time varying similarity measure.

The role correlation algorithm will determine whether the two groups G_t and G_{t-1} are the same group

by heuristically computing the time-varying similarity measure and comparing against the pre-defined threshold. The group correlation algorithm operates as follows:

1. For each group G_t , identify G_t and G_{t-1} as the same group if i) G_{t-1} has the strongest time-varying similarity with G_t , among all the groups in P_{t-1} and ii) the average number of connections is at least within T^{hi} percent of the average number of connections of G_{t-1} .
2. For each group pair (G_t, G_{t-1}) that remain uncorrelated, decide whether G_t and G_{t-1} represent the same logical group based on how similar the connection patterns between G_t and its neighbor groups are to those between G_{t-1} and its neighbor groups.

Step 1 decides whether the two groups G_t and G_{t-1} are identical based on the time varying similarity measure. As in Section 4.2, we compute the similarity measure based on the average number of connections between the groups and their common neighbors. However, finding the common neighbor set between G_t and G_{t-1} is not trivial. This is because we cannot simply assume that a neighbor $h_t \in C(G_t)$ and a neighbor $h_{t-1} \in C(G_{t-1})$ are the same host even if they have the same host identifier. We use the following techniques to identify the common neighbor set:

- If a neighbor h_t of G_t shares the same host identifier with the neighbor h_{t-1} of G_{t-1} and both have been considered highly likely to be the same host (i.e. $h_t, h_{t-1} \in H_{same}$), we assume h_t is the neighbor to G_t in the same way as h_{t-1} is to G_{t-1} .
- For each neighbor pairs (h_t, h_{t-1}) that are not considered as highly likely to be the same host, we assume h_t is the neighbor to G_t in the same way as h_{t-1} is to G_{t-1} if and only if the following condition is true. The connection set size of h_{t-1} is within T^{hi} percent of that of h_t and no other neighbor of G_{t-1} has the connection set size closer to that of h_t .

The algorithm then computes the time-varying similarity measure between each neighbor pair (h_t, h_{t-1}) , which meets the aforementioned requirements, as the minimum of the average number of connections between h_t and G_t and between h_{t-1} and G_{t-1} . If the sum of the similarity measures for all common neighbor pairs within the bounds of the specified thresholds, the algorithm declares that groups G_t and G_{t-1} mean the same group.

6 Results

In this section, we evaluate the performance of the algorithms using traces gathered over a day at two corporate networks. We show that the algorithms operate well for both networks and examine the effects of user-defined thresholds on the results of the role classification algorithm.

We call the two test networks *Mazu* and *BigCompany*. Mazu is part of the corporate network at Mazu Networks, Inc., in Cambridge, MA. It consists of 110 hosts, including engineering workstations, several servers, and laptops. Mazu develops various software products in the area of network security and monitoring. The BigCompany network consists of 3638 hosts, including workstations, servers, and many IP phones. For privacy reasons, BigCompany must remain anonymous.

6.1 Effectiveness of the Grouping Algorithm

We evaluate the effectiveness of the role classification algorithm by comparing the groups formed by the algorithm against the logical roles that hosts play as determined by knowledgeable network administrators. For all the experiments, unless otherwise noted, we set user-defined thresholds, $S^{hi} = 80$, $S^{lo} = 55$, and $K^{hi} = 7$. We examine how these thresholds affect the results in later sections.

Figure 4 shows some of the groups formed by the role classification algorithm running on the Mazu data and configured with the default parameters. Each circle in the figure represents a group and lists its members and its connections with other groups. Where possible, we have indicated the logical role of each host, which we obtained by asking the Mazu network administrator. (Of course, this logical information was not used in constructing the grouping.)

Observe that the role classification algorithm placed almost all engineering (*eng*) machines in a single group, 85. Also note that the number of connections of an engineering host varies from 4 to 9. Similarly, most machines used by sales, management (*admin*) and operations (*ops*) were placed in a single group, 87. The largest group, 80, contains new machines and test machines in the lab.

However, four hosts that are identified as engineering machines are placed in group 87 rather than group 85. The reason is that these machines do not communicate with a set of hosts that engineering machines in group 90 communicate with. As shown in Figure 4, each engineering machine in group 90 has, on average, one connection with group 10, which consists of a Unix mail server, and one connection with group 6, which consists of a source revision control server (not shown in

the figure). On the other hand, almost every sales host in group 87 communicates with both the Microsoft Exchange server and the NT sever from group 71, but not with the Unix mail server nor the source revision control server. In fact, there are just two connections between group 87 and each of groups 6 and 10. The four engineering hosts in 87 had connection patterns very similar to those of sales hosts, so they were grouped accordingly. Most probably, these machines are used by engineering managers who do not perform engineering-related tasks such as coding, and use the Exchange mail server instead of the Unix mail server.

If we have the perfect knowledge of the logical structure of the network, we can use that knowledge to quantify the resulting quality of the groups produced by grouping algorithms. One simple yet effective metric used in the cluster validation literature [16, 12] is *Rand Statistic*, which is based on testing whether a pair of objects belongs to the same group as decided by the grouping algorithm and according to our knowledge. Let P and P^* be the partitionings of hosts produced by a grouping algorithm and based on our knowledge respectively. Let SS , SD , DS and DD be the numbers of host pairs that belong to the same group in both P^* and P , to the same group in P^* and to different groups in P , to different groups in P^* and to the same group in P , and to different groups in both P^* and P respectively. SD and DS are indicative of how different P is from P^* . $Rand\ Statistic\ R = \frac{SS+DD}{SS+SD+DS+DD}$ is between 0 and 1. The higher the value, the more similar P and P^* are.

For the Mazu network, we were able to ascertain the logical roles of all except 8 hosts. We worked closely with the Mazu network administrator to obtain P^* , the ideal partitioning of hosts. We find that the partitioning produced by the grouping algorithm (with default parameters) achieves $SS = 452$, $SD = 710$, $DS = 133$, $DD = 3856$ and $R = 0.8363$. This shows that the results of the grouping algorithm reflect to a high degree our intuitive notion of the underlying structure of the network. We note that the reason for having a relatively high SD is because the algorithm identifies subsets of hosts within large groups as separate groups. For instance, the grouping algorithm produces a few different groups, each containing a single *eng* machine instead of putting them in group 80 (not shown in the figure). This is because those *eng* machines have the total number of connections far greater than the average number of connections that a host in group 80 does. Such distinction may prove useful in certain situations.

Table 1 lists the five largest groups produced by running the grouping algorithm on the BigCompany network. Again, we relied on information generated by the network administrator to help us understand whether the groupings generated by the algorithm matched the log-

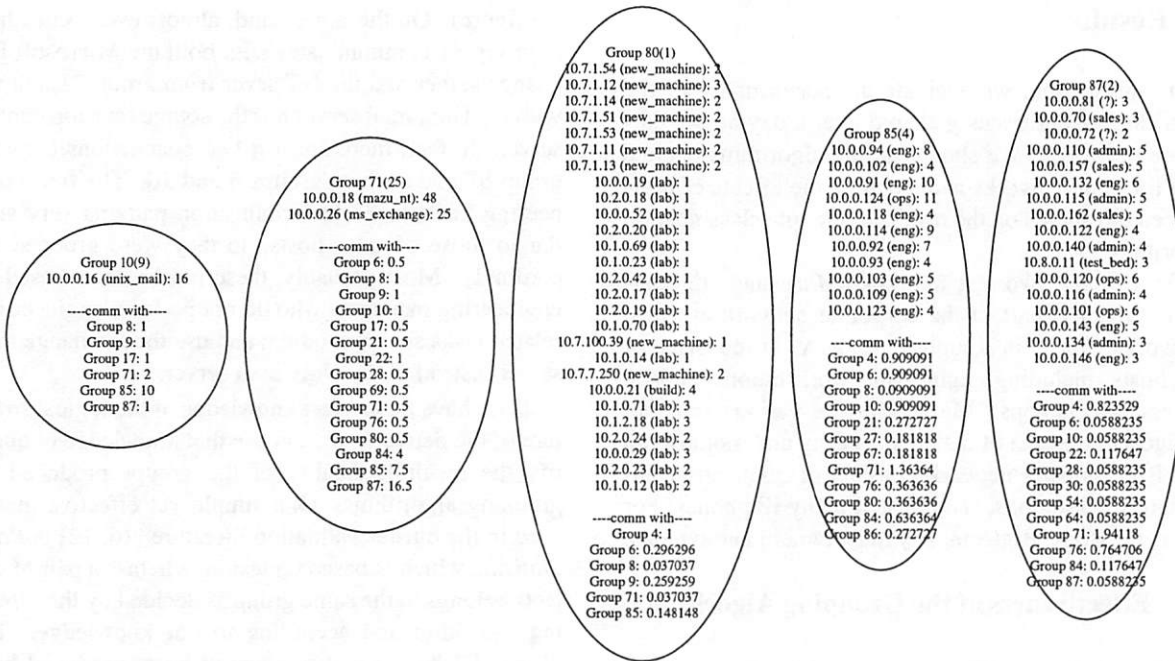


Figure 4. Grouping results based on data gathered over one day at Mazu. The number in parentheses next to the group ID is the group's K_G . The number next to each host is a count of the host's connections. Each line after "comm with" denotes a neighbor group and the average number of connections between the group and that neighbor.

ical structure of the network. Group 1020 consists of desktops whose IP addresses are managed by the DHCP server. Almost every machine in group 1020 communicates with approximately 85% of the machines in group 1075, and vice-versa. This pattern suggests that it was appropriate for the grouping algorithm to combine the machines in group 1075, which use static IPs, into a group. Most machines in both groups run Microsoft Windows. The high number of connections between the groups is due to Windows file sharing, which uses the NetBIOS network protocol. File sharing creates a large number of connections between the hosts in the two groups, even though in both groups there is little intra-group communication. We continue to investigate this interesting relationship between the two groups. It is striking, and further proof of the need for better analysis tools, that the network administrators we have talked to themselves don't know why the groups are partitioned in this way.

The grouping algorithm also correctly classifies all IP phones into one group, 1092. Group 1138 consists of web servers and other servers that desktops in group 1020 regularly communicate with. Group 1043 is the largest group, with 1519 members. Most machines in this group have a single connection (hence the role name

idle), and that is to a host that opens connections to about 1,600 machines. With our help, BigCompany is currently investigating why this host scans about 45% of all the machines on the network. This example is another good use of how the role classification algorithm can be applied to understand networks and detect anomalous behavior.

Table 2 summarizes the grouping results of the two networks. Observe that the number of groups in the BigCompany network is 26 times smaller than the number of hosts. Unfortunately, we cannot use Rand Statistic to quantify the quality of the groups produced by the grouping algorithm since we don't have the perfect knowledge of the logical roles of each machine in the BigCompany network. Nevertheless, the network administrators at BigCompany report that they find them both useful and consistent with their intuitions about their networks. We are also in the process of analyzing a larger network owned by HugeCompany that consists of 49,041 hosts.

6.2 Effectiveness of the Correlation Algorithm

This subsection shows that for a specific scenario, the role correlation algorithm associates new groups with

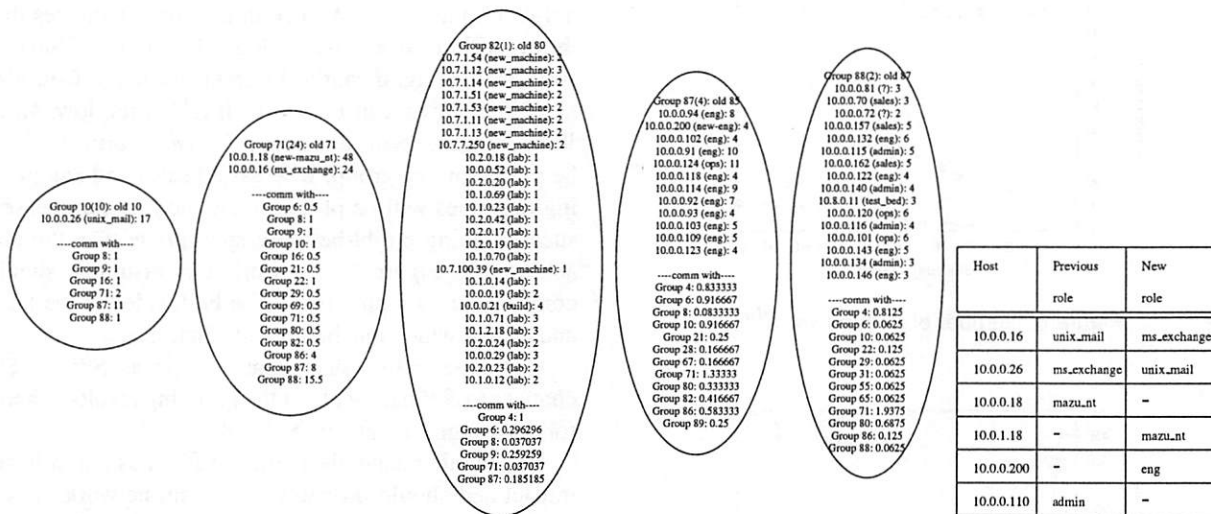


Figure 5. The grouping results on the Mazu network with several changes (see table) to the connection patterns. The number next to “old” represents the ID of the correlated group shown in Figure 4.

Group ID	Members	Logical Role
1043	1490	Idle
1020	158	DHCP-Desktops
1138	396	Servers
1092	167	IP-Phones
1075	156	StaticIP-Desktops

Table 1. The five largest groups classified in BigCompany network that consists of 3638 hosts. Logical role is identified by knowledgeable network administrators at BigCompany.

existing ones in an appropriate way. Figure 5 lists the scenario we investigate. In the Mazu network, we swapped the roles of *unix_mail* and *ms_exchange* by switching their IP addresses. We also replaced the old NT server, called *mazu_nt* (10.0.0.18), with a new server (10.0.1.18). Finally, we removed an old *admin* machine (10.0.0.110) and brought in a new *eng* machine (10.0.0.200). Although the specific scenario is just one of many possible ones, it includes the types of changes that could happen in a real network.

The modified connection patterns were used as inputs to the role classification algorithm. The role correlation algorithm then correlated the new grouping results with the original results. Every group in the new results is correlated with an old group. Figure 5 depicts the four groups that are affected by the changes. Observe

how the member compositions of these four groups change from the ones in Figure 4. Both *unix_mail* and *ms_exchange* are correctly identified in the same fashion as in Figure 4 despite their role reversal. The new NT server (*new-nt_server*) appropriately takes the place of the old one. Similarly, a new *eng* host is grouped with other *eng* machines. Despite various changes to the connection patterns, the role correlation algorithm was able to correctly associate each new group with an existing one. We continue to investigate the limits of the role correlation algorithm under rigorous changes in connection patterns.

6.3 Configuration

The algorithms use two internal constants that we believe are not sensitive to particular network connection patterns. The group formation phase of the role classification algorithm (see Section 4.1) requires a constant $0 \leq \alpha \leq 1$ to keep a host h from forming groups with other hosts that have less than a fraction α of the number of connections that h has. The group merging phase keeps the two groups from merging if the average number of connections of a group is not within $0 \leq \beta \leq 1$ of the other's (see Figure 3).

We set $\alpha = 0.6$ and $\beta = 0.5$. Our experiments with both Mazu and BigCompany networks indicate that the default values work well on at least two rather different networks. We believe that, in general, it will not be necessary to adjust these constants. Nevertheless, we plan to expose these parameters to network administra-

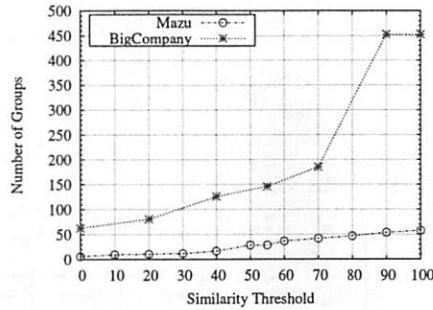


Figure 6. Number of Groups vs. S^{lo}

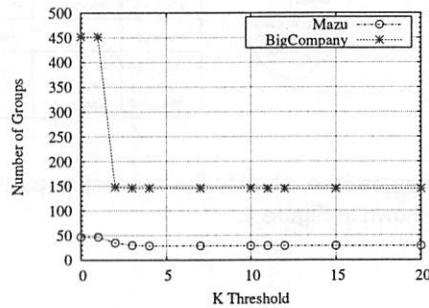


Figure 7. Number of Groups vs. K^{hi}

tors so that they can adjust them along with the similarity thresholds to achieve grouping results that most reflect their intuition of the network structure.

6.4 Effects of Similarity Thresholds

In this subsection, we examine how the choice of the user-defined thresholds, S^{lo} , S^{hi} , and K^{hi} , affect the number of groups formed by the role classification algorithm. Recall that the two groups are merged if and only if their similarity measure is $\geq S^{lo}$. Furthermore, if the maximum K_G associated with the groups is $\geq K^{hi}$, they are not merged unless their similarity measure is $\geq S^{hi}$. We require that $0 \leq S^{lo} < S^{hi} \leq 100$.

Figure 6 illustrates how S^{lo} affects the total number of groups formed for both Mazu and BigCompany networks. The number of groups increases with S^{lo} . Again, a large S^{lo} value keeps more groups from merging and as a result, the total number of groups remains large.

The number of groups may not increase smoothly with the increase in S^{lo} . For instance, there is steeper incline (knee) in the number of groups of BigCompany network when S^{lo} is increased from 70 to 90. The reason is that the increase in S^{lo} causes some groups with high numbers of connections to split, since they no longer meet the stronger similarity requirement to merge. This in turn causes several neighboring groups to

split. The extent to which such splits occur varies from network to network. A knee in the curve indicates that the algorithm can expose the logical structure of the network in two significantly different manners. Consider again the network in Figure 1. If S^{lo} is too low, *Mail*, *Web*, *SalesDatabase*, and *SourceRevisionControl* will all be placed in one group, whereas all sales and engineering machines will be placed in another. In some cases, such grouping might be more appropriate than the one achieved in Figure 1. Network administrators should compare the grouping results on both sides of the knee and decide which one better suits their needs.

Our experiments show that as long as $S^{hi} \geq 80$, changes to S^{hi} hardly affect the grouping results. Therefore, we suggest that S^{hi} be fixed.

On the other hand, the choice of K^{hi} has a significant impact and should probably vary from network to network. If K^{hi} is set to the maximum number of connections that any host has, the similarity measure between hosts is only compared against S^{lo} . If $K^{hi} = 0$, the similarity measure is only compared against S^{hi} . Ideally, K^{hi} should be set at a value that partitions the hosts in the network into two groups, one containing all server-like machines, and one containing all others.

Figure 7 shows how K^{hi} affects the number of groups formed. For any two data points with the same number of groups, the grouping results are identical. Clearly, the grouping results do not change for the Mazu network when $K^{hi} \geq 4$. Similarly, the grouping results hardly change for the BigCompany network when $K^{hi} \geq 3$. This implies that it is not too difficult to find an appropriate K^{hi} for a particular network. By default, we set $K^{hi} = 7$ and believe that this value will be suitable for most networks. Nevertheless, we are currently working on automatically setting K^{hi} .

6.5 Run Time

Table 2 shows the time taken to run the role classification algorithm on the Mazu and BigCompany networks. We performed our experiments on a Linux machine equipped with a 2GHz Intel Xeon processor and 4GB of memory. The run time achieved by the algorithms grows quadratically with the number of nodes and is acceptable for use in commercial enterprise network monitoring and analysis tools. We continue to further improve the performance of the algorithms.

7 Related Work

The work described in this paper was implemented in part using Click [21], a modular router system that makes it easy to build efficient packet processing devices on commodity PC hardware. The grouping al-

Network	Hosts	Groups	Run time(s)
Mazu	110	25	0.069
BigCompany	3638	137	63
HugeCompany	49041	1374	2101

Table 2. The summarized grouping results for Mazu and BigCompany networks.

gorithm only requires information about connections among hosts, so it can obtain data from a variety of sources, from summary formats like RMON [28] and Netflow [6] to packet-level sniffers like tcpdump [18].

Part of the role classification algorithm can be viewed as a data clustering algorithm. Data clustering has been an active area of research for a few decades [14, 19] and is known to be a difficult problem combinatorially. The techniques used to cluster data vary widely according to the assumptions, and contexts specific to application domains and many existing techniques are specifically developed for pattern recognition and image analysis. In general, a data clustering algorithm attempts to cluster data points or *patterns*, each of which is represented by a vector of real numbers. Patterns that are similar to each other are clustered together. The most popular metric for similarity measure is the Euclidean distance. One well-known clustering technique is the *hierarchical agglomerative clustering* technique. The idea is to merge clusters based on the pair-wise similarity measure of patterns. The merging process is stopped according to some predefined similarity thresholds. In this aspect, the group merging phase of the role classification algorithm can be classified as a hierarchical agglomerative clustering technique.

The main reason why traditional data clustering algorithms cannot be easily extended for our application domain is because it is difficult to represent the connection pattern of each host with a vector of numbers in such a way that the widely used Euclidean distance to measure the similarity between two connection patterns makes sense. Furthermore, we can leverage the communication patterns found in typical enterprise networks, such as client-server communications, to achieve more meaningful grouping results. We also note that traditional data clustering techniques do not deal with temporal correlation of clusters as the role correlation algorithm does.

The role classification algorithm is applicable to network intrusion detection. For example, grouping information provides context that can be used by intrusion detection systems [10, 22] (IDS) to help determine how unusual (and hence potentially suspicious) a certain network behavior is (see Section 2).

As explained in Section 2, role grouping is well-suited to improving network monitoring and policy management. An entire industry [8, 15, 17] caters to enterprises' network management needs, and much literature is devoted to network monitoring, traffic reporting, and performance measurement [13, 20, 23, 24]. All this work differs significantly from ours. The commercial network management systems are primarily integration and alerting tools, intended to provide operators with a unified view of disparate devices on the network. They serve as conduits for the raw data, but do not extract higher-level semantics such as role relationships. Academic work has focused on network monitoring and techniques for performance measurement, but again, the interpretation of data is generally left to humans.

Another tool that can help operators understand their networks is network visualization [1, 5, 11]. Visualization focuses on graphic design and automated layout algorithms to help users digest the vast amount of data generated by network monitoring tools. Unlike the grouping algorithm, these techniques have no notion of the logical structure of the network. However, they can complement grouping, exposing grouping information to the user and using grouping information to make better decisions about visual layout.

8 Summary

This paper has presented two practical algorithms (grouping and correlation) that group hosts on an enterprise network into roles according to their observed connection patterns. The first algorithm partitions hosts on the network into groups based on connection data. The second algorithm meaningfully correlates the results obtained by running the first algorithm at different times, taking into account the evolution of connection patterns over time.

To our knowledge, the problem of automatically grouping and classifying hosts based on their behavior on the network has not been addressed before. This paper formulates the problem by presenting an abstract model in addition to the concrete algorithm specifications. The general framework we have developed accommodates other classification algorithms in addition to the ones we have described.

Grouping hosts according to their connection habits exposes the logical structure of the network, and can serve to improve understanding of the network and to simplify a variety of network management tasks. It can also improve the accuracy of automated tools, such as systems for network monitoring and intrusion detection.

Experience with the algorithms on two corporate networks, one with about 100 hosts and one with over 3600 hosts, indicates that they work well. They are easy to

tune, and produce results that are meaningful and consistent with the intuition of experienced network administrators. Importantly, our experience on the corporate networks has shown that automated classification algorithms such as these can play an important role in assisting network administrators. The algorithms are also fairly efficient, and their performance remains practical even for networks with several thousand hosts.

Much work remains to be done. We plan to continue improving the performance of the algorithm. The ideal solution should be better than quadratic time complexity, since that could eventually be the limiting factor on very large networks. We will also explore other definitions of host similarity for grouping. For instance, one could consider incorporating services (such as TCP or UDP port information) or protocols into the definition of a connection, so that a web server would not be grouped with a mail server. In addition, we have yet to explore many of the applications of automatically-derived grouping information, which include network management, provisioning, security, and perhaps others.

References

- [1] BECKER, R. A., EICK, S. G., AND WILKS, A. R. Visualizing network data. *IEEE Transactions on Visualization and Computer Graphics* 1, 1 (1995), 16–28.
- [2] CERT COORDINATION CENTER. CERT Advisory CA-2001-19: Code Red Worm Exploiting Buffer Overflow In IIS Indexing Service DLL, July 2001.
- [3] CERT COORDINATION CENTER. CERT Advisory CA-2001-26: Nimda Worm. <http://www.cert.org/advisories/CA-2001-26.html>, September 2001.
- [4] CHESWICK, B., AND BELLOVIN, S. M. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison Wesley, 1994.
- [5] CHESWICK, B., BURCH, H., AND BRANIGAN, S. Mapping and visualizing the Internet. In *Proceedings of the 2000 USENIX Annual Technical Conference* (San Diego, CA, June 2000).
- [6] CISCO SYSTEMS. Cisco IOS NetFlow. <http://cisco.com/warp/public/732/netflow>.
- [7] CISCO SYSTEMS. Dynamic VLANs. http://cisco.com/univercd/cc/td/doc/product/lan/cat5000/rel_5_2/config/vmps.htm.
- [8] COMPUTER ASSOCIATES INTERNATIONAL. Unicenter. <http://cai.com/unicenter>.
- [9] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 1990.
- [10] DENNING, D. E. An intrusion-detection model. *IEEE Transactions on Software Engineering* 13, 2 (Feb. 1987), 222–232.
- [11] EICK, S. G., AND WILLS, G. J. Navigating large networks with hierarchies. In *Proceedings of the IEEE Conference in Visualization* (25–29 1993), pp. 204–210.
- [12] HALKIDI, M., BATISTAKIS, Y., AND VAZIRGIANNIS, M. On clustering validation techniques. *Journal of Intelligent Information Systems* 17, 2-3 (2001), 107–145.
- [13] HARRISON, H. E., MITCHELL, M. C., AND SHADDOCK, M. E. Pong: A flexible network services monitoring system. In *Proceedings of the USENIX 8th System Administration Conference (LISA VIII)* (San Diego, CA, Sept. 1994).
- [14] HARTIGAN, J. *Clustering Algorithms*. John Wiley, 1988.
- [15] HEWLETT-PACKARD COMPANY. OpenView. <http://openview.hp.com>.
- [16] HUBERT, L., AND ARABIE, P. Comparing Partitions. *Journal of Classification* 2 (1985), 193–218.
- [17] INTERNATIONAL BUSINESS MACHINES. Tivoli Software. <http://tivoli.com>.
- [18] JACOBSON, V., LERES, C., AND MCCANNE, S. tcpdump. <http://www.nrg.ee.lbl.gov>, 1991.
- [19] JAIN, A. K., MURTY, M. N., AND FLYNN, P. J. Data clustering: a review. *ACM Computing Surveys* 31, 3 (1999), 264–323.
- [20] KEYS, K., MOORE, D., KOGA, R., LAGACHE, E., TESCH, M., AND CLAFFY, K. The architecture of CoralReef: an Internet traffic monitoring software suite. In *PAM2001 Workshop on Passive and Active Measurements* (Apr. 2001), CAIDA. <http://www.caida.org/tools/measurement/coralreef/>.
- [21] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Transactions on Computer Systems* 18, 3 (August 2000), 263–297.
- [22] MUKHERJEE, B., HEBERLEIN, L., AND LEVITT, K. Network intrusion detection. *IEEE Network* 8, 3 (May-June 1994), 26–41.
- [23] PATARIN, S., AND MAKPANGOU, M. Pandora: a flexible network monitoring platform. In *Proceedings of the 2000 USENIX Annual Technical Conference* (San Diego, CA, June 2000).
- [24] PLONKA, D. Flowscan: A network traffic flow reporting and visualization tool. In *Proceedings of the USENIX 14th System Administration Conference (LISA XIV)* (New Orleans, LA, December 2000).
- [25] SIPSER, M. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
- [26] SNOEREN, A. C., AND BALAKRISHNAN, H. An End-to-End Approach to Host Mobility. In *Proceedings of the ACM Conference on Mobile Computing and Networking* (Boston, MA, 2000).
- [27] TARJAN, R. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.
- [28] WALDBUSSER, S. Remote Network Monitoring Management Information Base Version 2. Network Working Group RFC 2021, Jan. 1997.

A Cooperative Internet Backup Scheme

Mark Lillibridge

Sameh Elnikety

Andrew Birrell

Mike Burrows

Michael Isard

*HP Systems Research Center**

Palo Alto, CA

Abstract

We present a novel peer-to-peer backup technique that allows computers connected to the Internet to back up their data cooperatively: Each computer has a set of partner computers, which collectively hold its backup data. In return, it holds a part of each partner's backup data. By adding redundancy and distributing the backup data across many partners, a highly-reliable backup can be obtained in spite of the low reliability of the average Internet machine.

Because our scheme requires cooperation, it is potentially vulnerable to several novel attacks involving free riding (*e.g.*, holding a partner's data is costly, which tempts cheating) or disruption. We defend against these attacks using a number of new methods, including the use of periodic random challenges to ensure partners continue to hold data and the use of disk-space wasting to make cheating unprofitable. Results from an initial prototype show that our technique is feasible and very inexpensive: it appears to be one to two orders of magnitude cheaper than existing Internet backup services.

1 Introduction

Traditional data backup techniques work by writing backup data to removable media, which is then taken off-site to a secure location. For example, a server might write its backup data daily onto tape using an attached tape drive; at the end of each week, the resulting tapes would then be picked up by a truck and driven to a guarded warehouse. The main drawback of these techniques is the inconvenience for system owners of managing the media and transferring it off-site, especially for small installations and PC owners.

In contrast, Internet backup sites (*e.g.*, www.backuphelp.com), avoid this inconvenience by locating the tape or other media drive in the warehouse itself and by using the Internet instead of a truck to transfer the backup data. Customers need only install the supplied backup software to be assured that, so long as their system remains connected to the Internet, their data will be automatically backed up daily¹ without any further

action on their part. These sites charge by the month based on the amount of data being backed up. For example, a typical fee today to backup up one gigabyte of data is fifty US dollars a month (see Section 5.1).

In this paper we propose a new Internet-based backup technique that appears to be one to two orders of magnitude cheaper than existing Internet backup services. Instead of relying on a central warehouse holding removable media, we use a decentralized peer-to-peer scheme that stores backup data on the participating computers' hard drives.

To provide for off-site storage, we arrange for pairs of geographically-separated participating computers (*partners*) to swap equal amounts of disk space—a fair trade. To compensate for the fact that Internet PCs are much less reliable than a tape stored in a secure facility, we have each computer partner multiple times so it can spread its backup data in a redundant manner across many machines. By using a large number of partners per computer, we can ensure high reliability with low space overhead.

Our scheme requires the cooperation of the participating computers: computers depend on their partners to hold their data and make it available when needed. In an uncontrolled environment like the Internet, such cooperation cannot be taken for granted. Non-cooperation must be discouraged by making it unprofitable. We use several novel methods to do this, including the use of periodic random challenges to ensure partners continue to hold data (partners that fail are abandoned in favor of new partners) and the use of disk-space wasting to make fake crashes unprofitable.

The remainder of this paper is organized as follows: Section 2 describes a simplified version of our scheme that assumes cooperation can be taken for granted. It is well suited for systems that are intended to be deployed within a single company. Section 3 tells how to extend the simplified scheme to an environment where cooperation cannot be assumed, such as the Internet, by adding various security mechanisms. Section 4 presents results from an initial prototype. Section 5 compares our system to existing Internet backup sites as well as traditional backup techniques. Section 6 covers related work. Finally, we present our concluding remarks in Section 7.

*Current affiliations: Elnikety, Rice University; Birrell, Burrows, Isard: Microsoft Research.

2 The simplified scheme

Each computer that wishes to participate in our backup scheme runs special software. Under the software's direction, these computers link up and form a peer-to-peer system over the Internet or a corporate intranet. The same software, performing the same functions, runs on each computer—except for a single external matching server (see Section 2.1), the system is decentralized and functionally symmetric. Like most peer-to-peer systems, computers are free to join or quit the system at any time.

Each participating computer has some number of backup partners. For example, A might have partners B , C , and D . Partnership is a symmetric relation: A is also a partner of B , C , and D . Partnership is not, however, transitive: B and C need not be partners, and in general B and C may share no partners other than A .

How many and which partners a given computer has varies over time. Computers start with zero partners on joining and quickly add enough partners to handle their current backup needs. As their backup needs change, they may want to add or remove partners. Partners may also be changed if an existing partner is found to be wanting (e.g., due to excessive downtime) or a new computer needs partners.

Each pair of partners agrees at partnership-formation time to an amount of storage to be swapped and a level of uptime (time that they are running and connected) that they must maintain. Different pairs may reach different agreements. Suppose A and B agree to swap s blocks. Then each must reserve s blocks of their local disk for use by each other. The software, running in the background, performs reads from and writes to this space on behalf of requests from the other partner.

2.1 Finding partners

We suggest using a simple central server to keep track of the computers in the system and their partner needs. Many other methods of finding partners are possible—for example, a Gnutella-like flooding approach could be used—but the central-server method has the advantage of being very simple to implement.

Each computer should periodically update the server with its identity and what partners it needs and has, including uptime and storage-swapping levels. When a computer needs a new partner, it contacts the server with its needs and obtains a list of candidate partners; it can then contact those computers directly and find out if they are still compatible.

Sometimes there may be no other computers looking for new partners. In that case, a computer looking for new partners needs to step between two existing partners that have an agreement similar to the one it desires: if A and B are partners, N can step between them so

that A now has N for a partner instead of B and B now has N as a partner instead of A . This leaves A and B with the same number of partners, but gives N two new partners of the type it wants. By having N copy A and B 's data beforehand, this can be done atomically with no data loss.

To avoid complicated negotiation, we suggest appropriately quantizing uptime and storage-swapping levels. Ideally, to work well, the system should have many (at least a hundred, preferably more than ten thousand) members spanning the range of possible agreements. Computers wishing to swap huge amounts may still be out of luck finding compatible partners, but can compensate (with somewhat lower reliability) by using multiple partners swapping less each.

It is important that each partner in a pair be located at different sites in order to ensure all backups are stored off-site. Accordingly, computers should reject candidate partners that are co-located. This means that our scheme cannot be used safely within a single site. Additional reliability can be obtained by further diversification: a single computer should choose its partners from as many different sites and using as many different operating systems (to guard against viruses) as it can. To allow this, the identity information supplied to the central server should include a computer's "location" and operating-system type. Location information can either be obtained directly from the computer owner or estimated via IP ranges or domain-registry information.

Although the central server forms a single point of failure for finding new partners, it need keep no permanent state and is thus easily replaced or replicated should it fail or become a bottleneck. Its failure does not prevent backups or restorations from occurring; thus, as long as it is repaired within a reasonable amount of time (i.e., weeks), no real harm is done.

2.2 Creating a reliable logical disk

We use Reed-Solomon erasure-correcting codes [13] to create a highly-reliable logical disk from a large number of partners. A $(k+m, m)$ -Reed-Solomon erasure-correcting code generates m redundancy blocks from k data blocks in such a way that the original k data blocks can be reconstructed from any k of the $k+m$ data and redundancy blocks. By placing each of the $k+m$ blocks on a different partner, the logical disk can survive the failure of any m of the $k+m$ partners without any data loss, with a space overhead of m/k .

Erasure-correcting codes are more efficient than error-correcting codes because they handle only *erasures* (detectable losses of data) rather than the more general class of *errors* (arbitrary changes in data). Block errors can be turned into block erasures by attaching a checksum and version number to each stored block; all but the blocks

\mathcal{B}	\mathcal{C}	\mathcal{D}	\mathcal{E}	\mathcal{F}	\mathcal{G}
0	1	2	3	$R_{0,1,2,3}$	$R'_{0,1,2,3}$
4	5	6	7	$R_{4,5,6,7}$	$R'_{4,5,6,7}$
8	9	10	11	$R_{8,9,10,11}$	$R'_{8,9,10,11}$
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots

Figure 1: Sample block layout using 6 partners (\mathcal{B} - \mathcal{G}) with $k = 4$ and $m = 2$. $R_{w,x,y,z}$ denotes the first redundancy block and $R'_{w,x,y,z}$ the second redundancy block generated from data blocks w, x, y , and z .

k	m	n	Reliability	Overhead
6	0	6	53.144%	0%
6	1	7	85.031%	17%
6	2	8	96.191%	33%
6	3	9	99.167%	50%
6	4	10	99.837%	67%
6	5	11	99.970%	83%
6	6	12	99.995%	100%

Figure 2: Reliability and overhead for increasing values of m , holding k constant at 6, and assuming an individual computer reliability of 90%.

with correct checksums and the highest version number are considered erased.²

So that we can use a small fixed block size, we stripe blocks across the partners using the erasure-correcting code. See Figure 1 for an example block layout for a computer \mathcal{A} with 6 partners that is using $k = 4$ and $m = 2$. We place corresponding outputs of the erasure-correcting code on the same partner (*e.g.*, \mathcal{F} holds all the first redundancy blocks in Figure 1) to make reconfiguration easier (see Section 2.6).

Each computer decides for itself how to tradeoff reliability against overhead by choosing values for k and m ; these choices in turn determine the number of partners $n = k + m$ it needs. To get a feel for how these tradeoffs work in practice, see Figures 2 and 3. Figure 2 shows how reliability rapidly and overhead slowly increase as the number of redundancy blocks (m) is increased while holding the number of data blocks (k) constant. Figure 3 shows that the overhead can be decreased for a given level of reliability by increasing the number of partners ($n = k + m$). (Unlike traditional RAID systems, we can use high values of m and n because backup and restoration are relatively insensitive to latency.)

These figures were calculated via the binomial distribution assuming that individual Internet computers fail independently and are 90% reliable. More precisely, they assume that when a computer tries to restore its data, the probability that a particular one of its partners still has its data, uncorrupted, and is sufficiently avail-

k	m	n	Reliability	Overhead
6	6	12	99.995%	100%
8	7	15	99.997%	88%
10	8	18	99.998%	80%
12	9	21	99.999%	75%
14	9	23	99.997%	64%
16	10	26	99.999%	63%
18	10	28	99.996%	56%

Figure 3: Reliability and overhead for increasing values of k , using the minimum value of m necessary to get a reliability of at least 99.995% and assuming an individual computer reliability of 90%.

able during a limited restoration time window to supply that data is 90%.

This number is meant to be conservative; we expect from our personal experience that the real number will be considerably higher. Indeed, the only empirical study we know of on PC availability, Bolosky *et al.* [3], found that over half of all Microsoft's corporate desktops were up over 95% of the time when pinged hourly. As that study included some PCs that are shut down at night or over the weekend and a reasonable restoration window would probably be at least 24 hours, their numbers underestimate restoration availability for machines promising 24-hour availability. Nonetheless, even when using our conservative number, our calculations show that high reliability can be achieved with low overhead.

We expect randomly-chosen Internet PCs to fail independently except in cases of widespread virus damage, sustained loss of Internet connectivity, and (in the case of uncooperative environments) coordinated attacks involving multiple computers. See Section 3.3 for discussion of why we believe the later is unlikely to be a problem in practice.

2.3 Backing up data

Each computer backs up its data on the reliable logical disk it has constructed from its partners' disks. Exactly how this is done—*e.g.*, incrementals vs. full backups, compression, ignoring caches and application binaries, *etc.*—is orthogonal to our scheme; we assume here only that backups occur at most daily. Because our backup space is of limited size and somewhat expensive compared to removable media, it may be useful to conserve space. In particular, one may not want to require backup space for two full snapshots so that a crash while writing a new snapshot does not leave the system without a viable backup. We demonstrate one way of doing this in Section 4.1.

The following procedure can be used to stream a snapshot to logical disk starting on a block stripe boundary: For each k data blocks of the stream, perform the follow-

ing operations in turn: use the erasure-correcting code to generate the m redundancy blocks from the data blocks, generate and attach a checksum and the same new version number to each of the $k+m$ blocks, send a request to write each block to the appropriate partner in the appropriate place, and, finally wait for acknowledgments from at least $w \geq \max(k, m+1)$ partners. It is important that we write at least k blocks to the current block stripe before starting to write the next one to ensure that a crash while writing will leave at most one block stripe unreadable; we need to write at least $m+1$ blocks to ensure that the old version is overwritten.

The parameter w here represents a tradeoff. Smaller values of w allow the backup to proceed faster because it is not necessary to wait for as many partners to be up, but the resulting written data has lower reliability than normal because some of the blocks are missing: only $w-k$ failures can be tolerated before some of the written data is unrecoverable. w should be chosen based on empirical data and the uptime-level agreements being used. Note that if more than $n-w$ partners fail, it will no longer be possible to make new backups with this procedure until some of the failing partners have been replaced. Alternatively, w could be updated based on the number of partners scheduled to be replaced.

We run a *cleaner* in the background on each computer to help limit how long recently written data has less than the maximum redundancy available. The cleaner scans its logical disk looking for incompletely-written block stripes. Each time it finds such a block stripe, it reads as many blocks as it can from it and tries to decode the stripe. If it succeeds, it generates the missing blocks and writes them to the appropriate partners, thus increasing the stripe's redundancy. Note that both the cleaner and streaming procedure use only a block stripe worth of extra local storage, avoiding the need for an extra snapshot worth of temporary disk space during backing up.

2.4 Restoring data

Restoration can be done from any computer in the event of the backed-up machine's total destruction. The backed-up computer's logical disk can be recovered to the new computer's local disk given a list of the original computer's current partners by using the following procedure: Contact each partner and ask for all of the backed-up computer's data. For each block stripe, attempt to decode using the erasure-correcting code the blocks with valid checksums and the highest version number in that stripe. If you succeed, write the resulting data blocks to local disk in the appropriate places. Keep repeating this process, retrying partners that were down, until additional blocks cannot result in more stripes being successfully decoded or time runs out.

It is the responsibility of the backed-up-computer maintainer to keep one or more copies of the list of current partners off-site in a security box or the like. This list is generated shortly after joining once the initial set of partners has been determined and updated occasionally as partners change.

To limit how often this list must be updated, we store the list of current partners in a special block (the *master block*) that is replicated on each partner and not part of any block stripe. This means that the list can be retrieved from any current partner so that the off-site list actually needs to be updated only every $k-1$ partner changes under the assumption that we must tolerate m partners failing. If this is still too frequent, it is possible to add many additional partners that we only swap master blocks with.

2.5 Handling downtime

In the real world computers are often unavailable: they may be connected via a dialup line or suffer from frequent soft failures (e.g., Windows crashes). Partners must agree on a level of required uptime (e.g., "up 90% of the time" or "up during California business hours").

Lower levels of partner uptime decrease performance: backups and restores take longer because the computer must wait for partners to become available. For example, if a machine's partners are up only during business hours and it crashes during the weekend, no restore will be available until Monday morning. Efficient backups require most partners to be up simultaneously during some period of the day. This limits the ability of computers with low and unpredictable uptime to participate in our scheme.

Agreements are subject to being broken. For the simplified scheme, we assume that owners are not out to take advantage of or hurt others. We do not, however, assume that owners are reliable about maintaining uptime agreements. Owners might forget to leave their computer on as much as planned, underestimate how often their machine crashes, or change their computer-usage policy without remembering to tell the backup-system software.

To guard against this, each computer keeps track of its uptime and warns its owner when it is failing to live up to its end of its agreement. For the simplified scheme, we assume this reminder is sufficient to make the owner take any needed steps to correct the problem. In the full scheme (see Section 3), we actively police agreements (both uptime and storage swapping) and abandon partners who fail to live up to their end of an agreement.

2.6 Resizing the amount of backup space

Consider a computer currently swapping s blocks with each of $n = k+m$ partners. In return for $n \times s$ blocks

of local disk, it has access to a logical disk of size $k \times s$ blocks. If it needs additional logical-disk space, it can either add more partners swapping s blocks each (presumably maintaining a similar ratio of k and m) or switch to n new partners willing to swap $s' > s$ blocks each. Adding partners increases the amount of overhead due to per-partner costs (especially under the full scheme where we must periodically check on each partner), but requires issuing a new current partner list less often.

The same methods run in reverse can be used to shrink the logical disk. Partners holding redundancy blocks can also be added or removed to adjust the reliability level. Most of these changes require moving data around to maintain a sequential image (*i.e.*, adding partners adds blocks to every stripe row, rather than just adding a bunch of blocks at the end of the disk). By using the master block and version numbers, this can be done using no extra space in a restartable way with restoration always possible.

3 Security

In the previous section, we described a simplified scheme that assumes system members can be relied on to cooperate with each other, either because of substantially-similar interests or some external enforcement regime. We believe this assumption is likely to hold for systems deployed within a single company. Care should be taken, however, if our scheme is used within a single company to ensure sufficient site diversity so that all partnerships can be between sites.

In this section we describe how to extend the simplified scheme so that it can function in an environment such as the Internet where cooperation cannot be assumed because computer owners have different and possibly conflicting interests. Systems operating in such environments must be able to defend against members attempting to read or alter other members' data, to unfairly take advantage of other members, and to shut down or impair the system.

3.1 Confidentiality and integrity

To ensure the confidentiality of its backup data, each computer should encrypt its data before sending it to its partners using symmetric cryptography with a secret key known only to it. Because this and the other keys described below are needed for restoration, they should be added to the current-partners list that is manually taken off-site.

Ensuring backup integrity requires three steps. First, third parties must be prevented from impersonating a computer to one of its partners so that they can overwrite that computer's data. This requires pairwise authentication. At partnership formation time, the two partners can use Diffie-Hellman to establish a shared secret key,

which they can then use later to authenticate write messages by attaching a sequence number and keyed cryptographic hash to each message.

Second, partners must be stopped from modifying a computer's data by altering a block's data then fixing up its checksum. This can be prevented by substituting a keyed cryptographic hash for the simple checksum used by the simplified scheme. So long as a computer keeps this hash key (an *integrity key*) secret, no other party will be able to modify or generate new valid blocks. Like with the encryption key, there is no need to have separate integrity keys for each partner.

Third, the ability of a computer's partners to conspire to replace one valid block with another must be limited. Computing a block's cryptographic hash over the partner ID and block offset where that block is stored in addition to its portion of the backup data and version number will prevent all substitutions except those involving an earlier version of the same logical-disk block. By storing the date and version number of each snapshot in the master block and refusing to accept earlier versions at restoration time, a computer can ensure that conspirators can not selectively revert parts of a snapshot. A conspiracy of at least k partners can still revert the entire snapshot to a previous version; the only possible defense is to print the date of the actual snapshot being restored in the hope that the owner will notice the reversion.

The order in which encryption and checksum attachment are done matters. The correct order is to (1) generate the redundancy blocks, (2) encrypt each block, (3) attach the version number, (4) compute a cryptographic hash for each block (over the encrypted data, version number, partner ID, and block offset), and (5) attach the appropriate cryptographic hash to each block. This order allows a block's validity to be checked without having to decrypt it. More importantly, it ensures that there is no exploitable redundancy available to attackers: if encryption was done before generating redundancy blocks, a partner could save space by using the erasure-correcting code to reconstruct the data he was supposed to store from the data stored at the other partners. While good for him, that leaves the backup with lower redundancy.

3.2 Free-rider attacks

Peer-to-peer systems, including ours, are potentially vulnerable to *free-rider* attacks. A free-rider attack is one where an attacker, called a free rider, benefits from the system without contributing their fair share. The classic example of a free rider is a person who watches the US Public Television System (PBS) without donating any money. (PBS is supported largely by viewer donations.) Systems vulnerable to free-rider attacks either run at reduced capacity or collapse entirely because, as more and more users free ride, the costs of the system weigh more

and more heavily on the remaining honest users, encouraging them to either quit or free ride themselves.

3.2.1 Agreement violations

The most basic free-rider attack against our scheme is for an attacker to intentionally fail to uphold his end of his agreements. For example, under our simplified scheme a computer could free ride by letting its partners backup its data but refuse to hold their data in turn; this would give the computer backup service at essentially no cost to itself.

To prevent such attacks, participating computers should police their agreements by verifying whether or not their partners are honoring their promises. Each computer can periodically challenge each of their partners to make sure that the partner in question is up when promised and continuing to hold the data it agreed to hold.

Such a challenge might consist of a request for the block of the challenger's data stored at a challenger-chosen random offset; the answer would then be checked to make sure it is a valid block that belongs on that partner at that offset. Optionally, the block's version number could also be checked to make sure it is the most recent version. Because a partner who holds only fraction d of the challenger's data will pass c challenges with probability d^c , by challenging frequently enough, the challenger can be assured with high probability that its partner is still holding almost all of its data.

While a challenge remains pending (*i.e.*, not yet answered correctly), the challenger should keep retrying it until either it is answered correctly or the partner claims data loss (aka, needs restoration). After a computer has restored its data, it signals its partners, who then reload their data (empty blocks until their cleaners have a chance to run) on it and resume challenging it. The time the challenger spends waiting for an answer should be counted as downtime for that partner. Partners who are down too often (relative to their uptime-level agreement) or need restoration too often should be forever abandoned in favor of a new partner.

Unfortunately, a crashed computer looks just like a cheating one that is trying to dodge a challenge it cannot answer—both do not respond to requests. If computers using our scheme abandoned partners (discarding their backup data) as soon as they were clearly down more than their uptime agreements allow (say, 8 hours for a strict 100% agreement), our backup service would not be very useful because backups would likely be discarded before computer owners realized they were needed.

Accordingly, we suggest allowing a *grace period* of two weeks: partners should not be abandoned until two weeks have passed since they first went down excessively. We recommend two weeks to cover the case

where a machine crashes, losing data, just after the owner leaves on a two week vacation.

3.2.2 Exploiting the grace period

A more sophisticated free-rider attack involves taking advantage of the grace period to obtain backup service for free. The attacker joins the system, forming partnerships and exchanging data as normal. He then pretends to crash, throwing away all the data he has been given. For the next 2 weeks, he has free backup service because of the grace period. Just before the end of the grace period, when his partners will stop giving him backup service, he switches to a new set of unsuspecting partners and starts again. So long as he can find new partners (peer-to-peer systems can have millions of members), he can continue to receive backup service without cost.

Another free-rider attack involving the grace period has the attacker refusing to wait out the grace period before abandoning partners that are excessively down. This hurts his partners because they are left with a less redundant backup, or even no backup at all if enough of their partners free ride this way; however, the attacker benefits because his backup has better redundancy for the next two weeks because of the additional new partner.

The only way to deter these attacks is to make them unprofitable: we need to arrange things so that the attacker pays more for the privilege of using the grace period than it is worth to him and so that the attacker saves money by waiting out the grace period without abandoning his partners. (Free riders are motivated by the chance to save money, not the opportunity to hurt others.)

Payment If our scheme was modified to use a hybrid peer-to-peer model where a company charged money to use the software, collecting payment would be easy: just add an extra amount to the monthly bill. Likewise, if low-cost electronic financial transactions were available (*e.g.*, digital cash), payment could consist of a straightforward transfer of money between the attacker and the hurt parties or a suitable charity.

Because we wish to allow for a decentralized peer-to-peer system with almost no administration beyond keeping the matching server running and do not believe low-cost transactions are likely to become available anytime soon, we consider here a different way to make attackers pay: disk-space wasting.

The idea behind disk-space wasting is that we need to impose a cost that balances out the benefit the attacker stands to gain, in this case 2 weeks of a partner holding s blocks of his data. How much is this benefit worth? Clearly, no more than it would cost to obtain it from the next cheapest source, namely our scheme used honestly: the effort required to host s blocks for 2 weeks. Thus forcing someone to waste s blocks of disk space for 2

weeks cancels any benefit they might receive from abusing the grace period.

Assuming computers \mathcal{A} and \mathcal{B} are already swapping s blocks, \mathcal{A} can pay such a cost to \mathcal{B} by holding an additional s blocks for \mathcal{B} for two weeks, by holding an additional $2 \times s$ blocks for one week, or by allowing \mathcal{B} to hold s fewer of \mathcal{A} 's blocks for two weeks. The previously described protocols are used to allow \mathcal{B} to read, write, and check on her additional blocks. Note that in the fewer blocks case that \mathcal{B} is no longer holding \mathcal{A} 's backup data for the duration, leaving \mathcal{A} with one fewer effective backup partner. Also notice that unlike the monetary-payment cases, disk-space payments take time to make—one or two weeks in the examples here.

Such payments may benefit \mathcal{B} because she has more storage available to her, and thus represent a transfer of value from \mathcal{A} to \mathcal{B} . A different kind of payment is a *commitment-cost* payment where \mathcal{B} ensures that \mathcal{A} either pays an unrelated third party (e.g., a charity in the monetary-payment case) or else simply wastes resources (the disk-space-wasting case). We can convert the previous transfer examples into commitment-cost payments by using *low-utility blocks*.

A low-utility block is a normal block whose access has been sufficiently restricted so that it is of little or no utility to its lessee. For example, the lessee might be allowed to write any time, but be able to read only an occasional random one of its low-utility blocks for checking purposes. More draconian would be allowing only reads of random *words*. A transfer can be turned into a commitment-cost payment by either making the additional blocks held be low-utility blocks or by instead of having the partner hold s fewer normal blocks, have her convert those s normal blocks to low-utility blocks for the duration. The low-utility-block lessee stores uncompressible (i.e., encrypted) data in those blocks and checks a random block (or word) periodically to ensure that the data is being kept. The parties must mutually agree on the random block (or word) to be read—see Section 3.3.1 for a protocol to do this—to guard against the lessor always presenting the same block for inspection.

Because we believe that most PCs will not have enough space to hold a large number of extra blocks beyond the ones already needed for backup space, we will assume for the rest of this paper that the “allow your partner to hold s less normal blocks” cases are used for disk-space wasting. While this requires no extra space, it does have the drawback of leaving a PC with no backup while it is paying multiple partners. As an optimization, PCs paying commitment costs may continue to backup their data onto their (temporarily) low-utility blocks so that it is immediately accessible once payment is complete and the access restrictions are removed; this opti-

mization is incompatible with restricting access to random words because only entire backup blocks are verifiable.

Prepayment We can make these free-rider attacks unprofitable by requiring prepayment for the right to abuse the grace period: each time a computer gets a new partner, it pays a commitment cost of “3 weeks” to it and after being restored after d days of downtime, it must pay “ $d+1$ days” to its partners to keep them. Here, a cost of “1 day” is shorthand for disk-wasting for 1 day with the same amount of storage swapped or the equivalent via monetary transfers to a clarity or central billing authority. Note that if we do not use commitment-cost payments here, the payments between two new partners would cancel out.

This scheme clearly makes the backup-service-for-free attack unprofitable. The case for the refusing-to-wait-out-the-grace-period attack is more subtle: If the attacker switches immediately, he pays “3 weeks” for the new partner. If he waits instead, he might have to pay up to “2 weeks” for the grace period plus a possible additional “3 weeks” if the partner does not resume swapping data with him after restoration. So long as the probability of his partner resuming swapping is more than $2/3$, it will be cheaper for him to wait. Should the probability (q) turn out in practice to be less than this (unlikely), a larger new-partner fee of “2 weeks”/ q will still make the attack unprofitable.

The prepayment scheme has the advantages of being very simple and robust, requiring no assistance from the central server or any assumptions about the difficulties of changing computer identities. Its main disadvantage is that when disk-space wasting is used it interferes with backup service: backup service is not available for the first 3 weeks after joining the system, for up to 2 weeks after a restoration, and additional backup space takes 3 weeks to become available (new partners are needed). While growth in the backup space needed can usually be anticipated, the growth-speed limitation may be problematical in some cases.

Post-payment An alternative scheme is based on requiring payment *after* using the grace period (with or without a restoration). Here the central computer keeps track of each computer's partners. Each computer is supposed to honor the grace period and pay “ $d+1$ days” to its partners after being restored after d days of downtime. If it does not, its partners will complain to the central server, causing the server to sever those partnerships and impose a fine of “3 weeks” on all of the parties.

The fine must be imposed on everyone because, in general, there is no way for the central server to know who is truly at fault. This is unfortunate because it introduces a new free-rider attack: don't bother to complain,

letting others shoulder the burden of deterring attackers alone. We believe that this last attack will not be serious because it is only really tempting when there are a lot of attackers exploiting the grace period, which should not happen if most computers act to deter them by complaining.

The central server must impose the fine (e.g., it supplies the data and does the challenging) because an attacker's partners (new or old) may be accomplices that will not fine it. Should a computer refuse or try to cheat the fine, it is exiled by the central server from the system: the central server tells the machine's partners to abandon it and refuses to authorize any new partnerships for it ever again.

In order for the threat of exile to be an effective deterrent, rejoining the system under a new name must cost more than "3 weeks" times the maximum number of partners. A possible way to do this in a decentralized manner is by requiring joining computers to possess a class 2 personal digital certificate that has never been seen by the server before. Such certificates can currently be purchased from companies like GlobalSign (www.globalsign.net) for 16 Euros.

When disk-space wasting is used, this scheme provides better backup service availability than the prepayment one because it limits backup service only immediately after a restoration. It may, however, require the user to pay for membership somehow and requires much more effort from the central server per system member.

3.2.3 Bandwidth theft

Another free-rider attack involves attackers using participating computers to broadcast information. For example, a cracker might wish to make his pirated-software collection available to his friends but lacks the bandwidth to do this from his home PC. He can join our system and place the data to be broadcast on each of his partners in the clear; he then gives out his partners' IP addresses via email to his friends, who then download the data from the partners, subjecting them to unfair high traffic.

This attack is easily thwarted, however, by imposing a quota on how many reads or writes a partner can do per day, say three times the number needed for daily backup. A somewhat larger quota may be needed during restorations.

3.3 Disrupter attacks

Another kind of attack that we must guard against is *disrupter* attacks. A disrupter is an attacker motivated to disrupt, impair, or destroy a system or particular user. They might want to do this for prestige (any system whose disruption would make the front page is a target for some attackers) or to hurt a hated company or person.

1. \mathcal{A} chooses a random number $r1$
2. \mathcal{A} sends $Commit(\mathcal{A}, r1)$ to \mathcal{B}
3. \mathcal{B} chooses a random number $r2$
4. \mathcal{B} sends $r2$ to \mathcal{A}
5. \mathcal{A} sends opening information to \mathcal{B}
6. \mathcal{B} sends $block[r1 \oplus r2]$ to \mathcal{A}

Figure 4: The random-read protocol where \mathcal{A} is reading a block from \mathcal{B} . $Commit(-)$ generates a nonmalleable commitment that is revealed in step 5; it hides $r1$ from \mathcal{B} until step 5 while preventing \mathcal{A} from changing her mind after seeing $r2$.

Unlike free riders, disrupters cannot be deterred simply by making such attacks unprofitable; the attacks must be made ineffective or beyond the attackers' budget.

3.3.1 Blocking restoration systemwide

The basic disrupter attack against our scheme is to attempt to block restoration of as many machines as possible by controlling enough of their partners: any machine which has more than m attacker machines as partners will believe its backups succeed, but come restoration time will find that it is unable to recover its data because the attacker machines refuse to cooperate.

In order to attack a large number of machines (presumably for publicity) this way, the attacker will have to swap disk space with thousands of machines. The disk storage to do this directly, however, is beyond the budget of any likely attacker. However, by using a man-in-the-middle attack, a smart attacker can appear to be storing data for thousands of machines without using any local disk space. He simply steps between pairs of partners (see Section 2.1) and passes blocks back and forth, leaving each original partner to believe the attacker is backing up its data, when in fact they are still storing each other's data. If the attacker encrypts data going one way and decrypts data going the other way, he can ensure that when he bows out the stored data will be useless to the original partners.

We can prevent such man-in-the-middle attacks by changing our block-access protocols slightly. Intuitively, the idea is to provide only the following read operation: read a completely-random block. Unlike the normal read-specified-block operation, this operation cannot be passed through: if \mathcal{A} does a random read (say block r is chosen) of \mathcal{B} , there is no way for \mathcal{B} to read block r from \mathcal{C} efficiently because each random read he does has only a $1/s$ chance of returning block r . On average \mathcal{B} must read $s/2$ blocks to fulfill each random-read request of \mathcal{A} . This means that a small number of challenges by \mathcal{A} will quickly force \mathcal{B} to exhaust his read quota with \mathcal{C} (see Section 3.2.3), leaving him unable to answer all of the challenges.

This restriction, of course, must be lifted while a restoration is in progress and may be lifted systemwide periodically so that cleaners can run efficiently. Figure 4 shows one way to formalize the random-read operation into a protocol. Steps 1–5 here, modulo the inclusion of \mathcal{A} 's name in the commitment, are a standard variant of the flipping-coins-by-telephone protocol [2]; step 6 returns the resulting chosen block, the one at offset $r1$ xor $r2$.

We include \mathcal{A} 's name—either \mathcal{A} 's current IP address, or if that is spoofable, a public key—in the commitment in step 2 to prevent the random-number-choosing protocol itself from being passed through: If \mathcal{B} simply passes the commitment along to \mathcal{C} , he will be caught after he reveals the commitment in step 5 and \mathcal{C} sees that it does not have \mathcal{B} 's name in it. He is prevented from fixing up the commitment to have his name while still containing $r1$ by the non-malleability of the commitment scheme [9]. Because he is unable to pass through the choice of $r1$, he is unable to trick \mathcal{A} and \mathcal{C} into agreeing on the same random number.

3.3.2 Blocking one machine's restoration

The preceding defense prevents attackers from disrupting a large portion of our system; it does nothing, however, to prevent attackers from targeting one or two hated machines. It is not clear that much can be done to defend against such a focused attack. By requiring that all a computer's partners be located on different IP subnets, forcing partners to be chosen randomly from the eligible subset, and charging a fee for (excessive) switching the cost of such an attack could be raised somewhat, but probably not enough to deter determined attackers.

The attack is unlikely to be used, however, because it is not very effective at damaging the target machine compared to alternative attacks such as targeted viruses and denial-of-service attacks: it only blocks restoration; actual data loss requires an independent event that might take years to happen, if it happens at all. There is little point in combining this attack with a method for destroying a computer's data because it is usually easier to modify such methods to destroy any backup directly by corrupting the backup software.

4 The Prototype

We are building a prototype using our scheme, but it is not yet fully operational. Enough functionality is working, however, to allow measurement of the prototype's backup performance, which we report on here along with some interesting aspects of the prototype's data-layout choices.

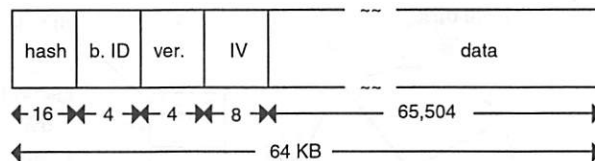


Figure 5: The format for blocks stored at partners, which includes a 16-byte HMAC-MD5 cryptographic hash, a block ID, a version number, an initialization vector (encryption-added random padding), and 65,504 bytes of backup data.

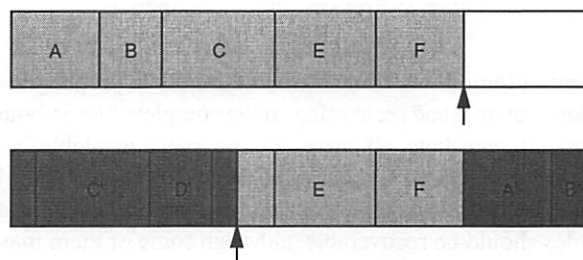


Figure 6: Sample partial overwrite of old snapshot (light grey) by new one (dark grey). Files A, B, and C are overwritten with new versions (A', B', C') and a new file D' is written. Only part of the new E is written, but it and F have old versions that are not yet overwritten.

4.1 Data layout

The prototype divides the logical disk into 64 KB blocks. Figure 5 shows the format used for these blocks. The necessary header fields use 32 bytes, leaving 65,504 bytes free for data, an overhead of only 0.05%. The prototype uses the IDEA block cipher for encrypting the data and the cryptographic-hash HMAC MD5 to generate checksums in the order described in Section 3.1.

The prototype treats the logical disk as if it were a large circular tape: each snapshot is written starting just after the last one, using ascending block offsets (with wrap around at the end of the disk). Snapshots use a format similar to archival file formats (e.g., tar), which store a sequence of files by writing, for each file, a file header followed by that file's data. The file header contains a synchronizing sequence, the file name, date stamp, file length, and a checksum. Because of the synchronizing sequence, it is possible to start reading a snapshot in the middle and still extract all the files whose headers come after that point.

This property of the snapshot format can be useful when backup space is limited, and as a result the next snapshot necessarily overwrites the last full snapshot: Should the computer crash while writing the new snapshot, it will be left with two incomplete snapshots at restoration time, the beginning of the new one and the

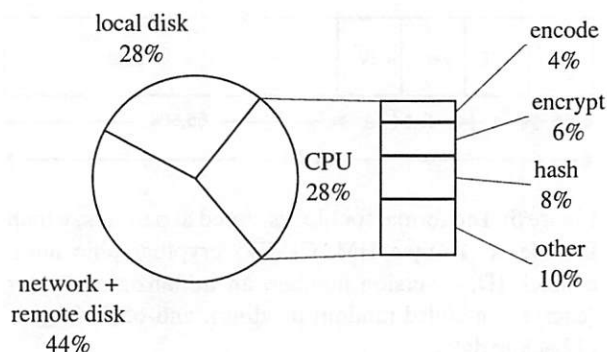


Figure 7: Breakdown of backup time.

end of the old one. The start-in-the-middle property allows reading and recovering all the complete files in both partial snapshots. If there is extra space available beyond that needed to hold a full snapshot and the set of files being backed up has not changed greatly, then most files should be recoverable, although some of them may be restored to the version saved in the old snapshot; see Figure 6 for an example.

4.2 Performance

To measure the prototype's backup performance, we used five personal computers running Microsoft Windows NT, each with a 200 Mhz Pentium Pro processor, 64 MB of RAM, and a 10 Mbps Ethernet card. The computers were connected via a 10 Mbps Ethernet hub. We used the (8,2)-Reed-Solomon erasure-correcting code, which tolerates the failure of any 2 of 8 partners at the cost of using $2/6 = 33\%$ extra space, and simulated 10 participating computers by running two instances of the prototype software on each PC. Each instance was partnered with the eight instances located on different PCs from it, so that all communication between partners went through the network.

We instructed one partner to write either 100 MB or 1 GB of test data stored on its local disk to its logical disk to simulate saving a snapshot; during this time, the other partners were idle except for processing write requests. The prototype uses the procedure described in Section 2.3 with $w=7$ to write snapshots: writes to partners occur in parallel, but are not pipelined with the reading and preparing of the blocks to be written, and at most one write is in progress at each partner at a time.

Using this unoptimised procedure, the prototype is able to write 100 MB in 12 minutes and 1 GB in 2 hours. This corresponds to a backup rate of 1.0 Mbps and a write rate of 1.3 Mbps (larger because of the redundancy blocks that must be written). Optimization would improve these rates, but since they are already larger than the bandwidth many Internet connections provide, it is not clear how useful this would be.

space	100 MB	1 GB	10 GB	100 GB
mean	\$16.46	\$53.84	\$232.28	\$1773.33
min	\$4.50	\$12.71	\$72.21	\$720.00

Figure 8: Monthly fees in US dollars required by existing Internet Backup services to store a given amount of data; excludes one-time startup fees but includes discounts for annual contract.

Figure 7 breaks down the contributions to the backup time made by the local disk (mostly reads), the remote-writing step (mostly network and partner delays), and the various CPU-intensive tasks. The remote-writing step consumes the largest portion (44%) of the total time, presumably due to our allowing only one outstanding write to a given partner at a time. Hashing requires more time (8%) than encryption (6%) because it must be done twice for each block: once to generate the stored-block checksum and once to authenticate the write request containing that block to the partner. (Separate keys, and hence hashes, are required because the integrity key must be kept secret from the partner.)

5 Comparison with existing services

5.1 Cost

We did a survey of Internet backup sites on October 28, 2002. Figure 8 shows the average and minimum monthly fees for various amounts of storage for the 15 sites (out of 28 surveyed) that list prices on the web for given amounts of data to be stored (as opposed to the amount to be backed up, which differs due to assumptions about compression and how much data actually needs to be backed up). The cheapest marginal cost found was US \$7.20 per gigabyte.

In estimating the cost of using our scheme, we assume that the user's computer hardware, base power, and any needed bandwidth are already paid for by existing uses. This seems reasonable for a home PC that is already on enough for reasonable predictable uptime and that has an Internet connection with flat-rate bandwidth pricing. We furthermore assume no cost for the software or central-server operation based on an open-source model and the extremely low overhead on the central server per participating computer; if a commercial model of software development was used instead, a small one-time fee for the software might be also be required. Under these assumptions, the cost of our scheme is determined by the marginal cost of storage for a PC and the marginal cost of the extra power needed to operate a disk drive to answer challenges and backup data.

Based on the cost of a 60 GB internal IDE hard drive as of October 2002—US \$75 according to www.pricewatch.com—depreciated over 2 years, we conservatively estimate the marginal cost of storage

at no more than 5.2 US cents per gigabyte per month. The cost of the extra power required is much harder to estimate. One conservative approach is to assume that the disk is turned on 1.5 extra hours per day per gigabyte to be backed up, on the assumption that the average backup takes 25% of the time of a full one (2 hours/GB for the prototype in each direction) and that challenges take half an hour of disk time total per day. Desktop disk drives appear to consume about 10 watts extra power when active [11] so at a conservative electricity cost of 15 cents per kilowatt hour, our scheme should use 7.5 US cents of power per GB backed up per month.

If we assume 100% storage overhead for redundancy (e.g., $k=m$) and room for 2 full snapshots on the logical disk (a factor of 2.2 should suffice), we will need to trade 4.4 GB of local disk in order to back up 1 GB of data, resulting in our scheme costing no more than 26 US cents per gigabyte per month. A less conservative estimate (3 years, 50% overhead) gives a figure of 18.6 US cents/GB/month.

Thus, our scheme appears to be 30 to 100 times cheaper than existing Internet backup services. We do not fully understand why this is, but believe it largely stems from our scheme's lack of administrative costs and use of marginal resources (e.g., most of the resources we use are already paid for by other uses). A small part of the difference may be due to limitations of our scheme as compared to existing Internet backup services (see below).

Traditional backup methods can be comparable in cost to our scheme, but are inconvenient for users: if a home-computer user weekly writes a snapshot to 700 MB CD-Rs (6 US cents each in quantities of 100) then takes them to work and leaves them there, he will incur a cost of 35.1 US cents/GB/month ignoring the cost of the CD burner.

5.2 Limitations

Our scheme does have some limitations as compared to existing Internet backup services. Perhaps the biggest limitations are the limited grace period (2 weeks) and the need for sizable amounts of predictable Internet connectivity. Unlike the existing services, which provide long grace periods (months if not years) and only require a computer to be connected to the Internet when a snapshot needs to be saved, our scheme leaves computers with no backup at all in case of excessive downtime. Unlike schemes based on off-line media, our scheme offers little protection against catastrophic viruses that suddenly erase most PCs' hard drives.

Also, our scheme is somewhat less convenient than the best Internet backup services: we provide no tech support line to hold users hands and will not FedEx a CD copy of a user's data to them so they don't have to

wait for a restore over a slow Internet connection. Depending on the uptime-level agreement, restoration may take longer in our scheme than with the existing services. In theory, Internet backup sites could offer insurance ("if we lose your data, we'll pay you a million dollars"). Insuring users in our system seems problematic due to possible fraud using the disrupter attack of Section 3.3.2.

If disk-space wasting is used, there are the additional limitations that backup service will not be available for 2 weeks after a crash, and if prepayment is used, backup service will not be available for the first 3 weeks and growing backup-storage space will take 3 weeks.

5.3 A hybrid model

It is possible to remove most of these limitations by combining our scheme with the existing Internet-backup-service model: A company would run the central server and bill users periodically. Backup would be done as per our scheme except that before abandoning a partner, a computer would upload the partner's data to the central site, where it would be stored temporarily. When the abandoned computer found a new partner, it would move its data from the central server to the new partner.

This model allows preserving backups even in the presence of excessive downtime. Members would be billed at a basic rate similar to our scheme but with surcharges for centrally storing data (charged presumably at the existing Internet-backup-services rate) and for abusing partners. In essence, customers with good uptime would pay our cheap rates while still being guaranteed a backup even if they exceed the 2 week grace period or are down excessively, albeit at a slightly higher price.

6 Related work

The first wave of peer-to-peer systems included several systems (The Eternity Service [1], Archival Inter-memory [4], Free Net [5], and Free Haven [7]) designed to provide *uncensorable storage*: documents once deposited in these systems cannot be altered or destroyed by attackers, even national governments. Like ours, these systems use cryptography and redundancy (Archival Inter-memory uses erasure-correcting codes) to protect data. It was soon suggested that one of these systems would make a good base on which to build a backup system.

Unfortunately, however, these systems do not have working defenses against free riders when used in this way: Freenet keeps only the most popularly requested documents, Archival Inter-memory simply assumes cooperation (it is intended for research libraries, which are believed to have substantially similar interests), and the Eternity Service paper only suggests a possible line of direction for a defense.

The Free Haven project has proposed an elaborate scheme based on trading storage, where nodes are allowed to insert only as many *shares* (the unit of trade in Free Haven) as they hold for others. However, unlike in our system, these shares are constantly traded between nodes, leading to effectively non-symmetric partnerships. This prevents a node *A* from directly punishing a node *B* that drops one of *A*'s shares by dropping one of *B*'s shares that *A* holds in return. Instead an elaborate reputation system is needed to punish nodes that are complained against too many times because they drop data. Unfortunately, building a working reputation system is very hard and it is far from clear if their system works [8].

More recent peer-to-peer storage systems include PAST [10, 14] and OceanStore [12, 15]. These systems do not attempt to provide uncensorability, and are thus simpler than the previous systems.

PAST relies on trusted third parties and smartcards to broker requests between clients so that clients cannot use more remote storage than they are providing locally. Unfortunately, insufficient details are provided about PAST's defenses to properly evaluate PAST's resistance to free-rider attacks—e.g., nodes are supposed to be randomly audited, but no details of how or with what consequences are provided. PAST appears to encrypt data before replicating, which may allow a free-rider attack where malicious nodes obtain data from the redundant data that their peers store, rather than storing it themselves (see Section 3.1).

OceanStore is a federated system where utility companies pool their resources to provide storage to users. Each user contracts with a single company, the *responsible party*, to receive storage for a fee. That company then exchanges storage with the other companies for greater reliability and geographic range. Because of the companies' large size and deep pockets, legal contracts and enforcement can be used to punish companies that do not keep their end of the bargain, based on planned billing and auditing systems. OceanStore, because of its need to support concurrent updates, is very complicated (e.g., it uses Byzantine agreement) and requires a great deal of central resources, making it likely more expensive than our scheme if only simple backup service is needed.

The only other peer-to-peer system we know of whose primary purpose is backup service is Pastiche [6]. Like in our scheme, Pastiche nodes form partnerships with other nodes. However, rather than using erasure-correcting codes, each Pastiche node stores a copy of all of its data on each of its partners. By sacrificing a fair amount of privacy (observers can tell if a node's filesystem includes a given large byte sequence), this backup data can be greatly compressed: by choosing partners with similar software installations, most files

being backed up will be already present on the partner, thus requiring no extra storage. No data is available yet about whether this savings compensates for the greater overhead of using full replication rather than erasure-correcting codes in practice. Pastiche has not yet adopted defenses against free-rider attacks; the authors merely sketch, in a paragraph each, three approaches that they are considering.

7 Conclusions

In this paper, we have described a scheme for a peer-to-peer Internet backup system that appears to be one to two orders of magnitude cheaper than existing Internet backup services. We believe the cost savings stem largely from savings on administrative expenses and the use of cheaper resources, much of whose operating cost is paid for by other uses. Preliminary experiments with a prototype show that the scheme's performance is acceptable in practice.

The most difficult part of the scheme design was guarding against the inevitable free rider and disrupter attacks to which a cooperative system is vulnerable. We came up with several novel mechanisms—periodic random-block challenges, disk-space wasting, and limiting reads to mutually-chosen random blocks—to address these attacks.

Acknowledgments

We wish to thank Marcos Aguilera, the anonymous referees, and our shepherd, Steve Gribble, for their valuable comments and suggestions.

Notes

1. Increased convenience encourages more frequent off-site backups.
2. This assumes all blocks (of a block stripe) are written as a unit; we do not discuss the more complicated partial-write case in this paper.

References

- [1] R. Anderson. The eternity service. In *Proceedings of Pragocrypt '96*, pages 242–252, Prague, Czech Republic, 1996. CTU Publishing House.
- [2] Manuel Blum. Coin flipping by telephone: A protocol for solving impossible problems. In *Advances in Cryptology: A Report on CRYPTO 81*, pages 11–15. Department of Electrical and Computer Engineering, U. C. Santa Barbara, August 1982. ECE Report 82-04.
- [3] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set

- of desktop PCs. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*, pages 34–43, Santa Clara, June 2000.
- [4] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the fourth ACM Conference on Digital libraries (DL '99)*, 1999.
 - [5] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability, LNCS 2009*, New York, 2001. Springer.
 - [6] L. P. Cox and B. D. Noble. Pastiche: making backup cheap and easy. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.
 - [7] Roger Dingledine. The free haven project: Design and deployment of an anonymous secure data haven. Master's thesis, MIT, June 2000.
 - [8] Roger Dingledine, Nick Mathewson, and Paul Syverson. Reputation in privacy enhancing technologies. In *Proceedings of Computers, Freedom, and Privacy*, April 2002.
 - [9] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.
 - [10] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of HotOS VIII*, May 2001.
 - [11] Keith Farkas. Personal communication, March 2003.
 - [12] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Nov. 2000.
 - [13] James S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software, Practice and Experience*, 27(9):995–1012, 1997.
 - [14] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of 18th ACM SOSP'01*, October 2001.
 - [15] Chris Wells. The OceanStore archive: Goals, structure, and self-repair. Master's thesis, U.C. Berkeley, May 2000.

Currentcy: A Unifying Abstraction for Expressing Energy Management Policies

Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat*

Department of Computer Science

Duke University

{zengh,carla,alvy,vahdat}@cs.duke.edu

Abstract

The global nature of energy creates challenges and opportunities for developing operating system policies to effectively manage energy consumption in battery-powered mobile/wireless devices. The proposed *currentcy model* creates the framework for the operating system to manage energy as a first-class resource. Furthermore, currentcy provides a powerful mechanism to formulate energy goals and to unify resource management policies across diverse competing applications and spanning device components with very different power characteristics.

This paper explores the ability of the currentcy model to capture more complex interactions and to express more mature energy goals than previously considered. We carry out this exploration in ECOSystem, an “energy-centric” Linux-based operating system. We extend ECOSystem to address four new goals: 1) reducing residual battery capacity at the end of the targeted battery lifetime when it is no longer required (e.g., recharging is available), 2) dynamic tracking of the energy needs of competing applications for more effective energy sharing, 3) reducing response time variation caused by limited energy availability, and 4) energy efficient disk management. Our results show that the currentcy model can express complex energy-related goals and behaviors, leading to more effective, unified management policies than those that develop from per-device approaches.

1 Introduction

Energy is an increasingly important system resource. This is most evident in battery-powered mobile computing platforms, from laptops to tiny embedded sen-

sor nodes, although its significance is becoming recognized in other computing environments as well. While a number of efforts have explored minimizing the power consumption of specific system resources (e.g., dynamic voltage scaling algorithms for the CPU, disk spindown policies, protocols using wireless power modes), recent work advocates that the operating system should explicitly manage the system-wide role that energy plays [17, 5] and view it as an opportunity and challenge for unifying resource management.

Our recent development of a framework for an energy centric operating system [22] proposes *currentcy* as a unifying abstraction for the management of a broad variety of system devices that consume energy. This work demonstrates the use of the currentcy model for expressing our overall battery lifetime goal and capturing the impact of individual system devices on battery lifetime. However, just as there is no single performance metric for all workloads, there is no single energy goal that satisfies all mobile/wireless scenarios. Thus, for this work, we set out to determine whether currentcy is general enough to express additional complex system behavior. Specifically, this paper makes the following contributions:

1. For some applications, it is important not just to achieve a target battery lifetime but to perform more work during that lifetime. Since characterizing “work” in general-purpose workloads is difficult, we look at fully utilizing the available energy within the specified time. Consider a sensor node running on rechargeable solar cells. The goal here might be to minimize residual energy remaining at sunrise after operating through the night (when it becomes possible to recharge – assuming reliable weather forecasts) to deliver maximum system utility. Any residual capacity at the end of the designated lifetime suggests overly conservative management and lost opportunities. Experience shows that this situation can result from a mismatch of the user specifications and actual demand. Thus, we translate this goal into the currentcy model and develop a currentcy conserving

*This work is supported in part by the National Science Foundation (EIA-99772879, ITR-0082914, CCR-0204367), Intel, and Microsoft. Vahdat is also supported by an NSF CAREER award (CCR-9984328). Additional information on this work is available at <http://www.cs.duke.edu/ari/millywatt/>.

energy allocation policy to reclaim unspent energy by adapting to observed energy consumption patterns.

2. Experience with the ECOSystem prototype also indicates that there can be subtle interactions between energy allocation and CPU scheduling policy. Scheduling that is oblivious to energy consumption may not provide adequate opportunity to spend currentcy allocations. Returning to our sensor example, suppose there is an important task that consumes most of its energy in the wireless network interface, communicating sensor readings, rather than on processing within the CPU. Such a task may experience a form of priority inversion. Thus, we develop a currentcy-based scheduling policy that recognizes the global relevance of energy consumption anywhere in the system on the scheduling decision. The result is more robust proportional sharing of energy regardless of which resources are favored by tasks.

3. Response time variability is disruptive in many applications. Whenever energy availability is constrained – which in ECOSystem means currentcy allocations are limited – it becomes important to have a steady rate of consumption. Thus, we develop a currentcy-based scheduling policy that achieves well-paced energy consumption, reducing response time variation.

4. For devices that have nontrivial transition costs between power states, such as a disk with spindown capability, there is potential for increased energy efficiency. We demonstrate how to shape disk access patterns to amortize the energy costs of spinup/spindown across multiple requests and thereby reduce the average energy used per request. We further show the energy and performance benefits of aggressive prefetching while the disk is spinning.

In summary, our experimental results show that the currentcy framework is successful in achieving more mature energy goals than previously pursued. These include: reducing residual energy, dynamically balancing per-task energy supply and demand, lowering response time variation, correcting energy-related scheduling inversions for improved energy sharing, and increased efficiency for disk accesses.

The rest of this paper is organized as follows. Section 2 describes the currentcy model and its implementation in ECOSystem, a Linux-based prototype, followed by a discussion of related work. Next, in Section 3, we outline several methods for manipulating currentcy to implement energy-related goals. Sections 4 through 8 describe the formulation of several energy goals, beyond simple battery lifetime, built upon the currentcy manage-

ment framework. We propose and evaluate solutions for each of these problems. Section 9 concludes this paper.

2 Background and Related Work

2.1 The Currentcy Model

The ECOSystem approach is based upon a unifying currentcy model. The key feature of this model is the use of a common unit—*currentcy*—for energy accounting and allocation across a variety of hardware components and tasks. Currentcy is an abstraction for explicitly representing energy as a resource, precisely specifying energy-related goals, and capturing the interactions among energy consumers in the system. It is the basis for characterizing the application power requirements and gaining access to any of the managed hardware resources. Currentcy is the mechanism for establishing a particular level of power consumption and for sharing the available energy among competing tasks.

Originally, the primary goal of ECOSystem was to achieve a target battery lifetime. Not only is this a well-defined metric to adopt as a starting point for explicitly managing energy, but there are also interesting application scenarios for which this is an appropriate objective. Exploiting battery properties that relate lifetime and discharge rate, ECOSystem expresses this goal in terms of the currentcy model and allocation strategies.

There are two levels to the energy allocation strategy. The first level allocation determines the amount of currentcy to collectively allocate among all tasks system-wide. We divide time into energy-epochs. At the start of each epoch, ECOSystem allocates a specific total amount of currentcy. For our original purpose, the overall currentcy allocation is determined by the discharge rate necessary to achieve the target battery lifetime. By distributing less than 100% of the currentcy required to drive a fully active system during an epoch, components are idled or throttled.

The second level of currentcy allocation is distribution among competing tasks. When the available currentcy is limited, it is divided among the competing tasks according to user-specified proportions. During each epoch, an allowance is granted to each task according to its specified proportional share of currentcy. There are constraints on the accumulation of unspent currentcy so that epochs of low demand do not amass a wealth of currentcy that could result in very high future power consumption peaks that would violate our battery assumptions. Consequently, there is a cap on the maximum amount of currentcy any individual application can save. Thus, the per-task allocation represents income in each epoch, whereas the cap represents a limit on the balance accumulated

within the task's account.

ECOSystem uses a reimplementa-tion of the Resource Containers [1] abstraction to capture the activity of an application or task as it consumes energy throughout the system. Resource containers are the abstraction to which currency allocations are granted and the entities to be debited for energy consumption. They are also the basis for proportional sharing of available energy. Resource Containers address variations in program structure that typically complicate accounting. For example, an application constructed of multiple processes can be represented by a single Resource Container for the purposes of energy accounting. We use the terms "task" and "resource container" interchangeably.

The energy accounting challenge of tracking energy use and attributing it to the responsible task is addressed through a power states model maintained within the framework. This model allows us to track interactions among tasks through their use of energy in device access. For example, tasks may be consuming energy in devices even when they are inactive in the CPU (or blocked). A process waiting for completion of a disk request is responsible for the energy consumption of the disk access. Ready-to-run processes may also be consuming energy in other devices (e.g., due to asynchronous I/O) while competing for the CPU.

The ECOSystem prototype [22] is a modified RedHat Linux version 2.4.0-test9 running on an IBM ThinkPad T20 laptop. This platform has a 655MHz PIII processor and we assume an active power consumption of 15.55W. The disk is an IBM travelstar that we model in ECOSystem with costs of 1.65mJ per block access and 6000mJ for both spinup and spindown, and with progressive costs/timeouts for levels of idle power states. The wireless network is an Orinoco Silver PC card supporting IEEE 802.11b, it has three power modes: Doze (0.045W), Receive (0.925W) and Transmit (1.425W). All other devices contribute to the base power consumption, measured to be 13W for the platform.

ECOSystem supports a simple interface to manually set the target battery lifetime and to prioritize among competing tasks. These values are translated into appropriate units for use with our currency model. The target battery lifetime is used to determine how much total currency can be allocated in each energy epoch. The task shares are used to distribute this available currency to the various tasks. To perform the per-epoch currency allocation, we introduce a new kernel thread *kenrgd* that wakes up periodically and distributes currency appropriately.

Initial experience and experiments with the prototype show that it can successfully deliver its goal of achieving a target battery lifetime and proportionally sharing available energy among competing applications using differ-

ent devices in the system. It has also identified some drawbacks including a disproportionate impact on performance. This work refines energy goals beyond the simple battery lifetime metric. The purpose of this paper is to explore the power of the currency model to express more subtle and sophisticated desired behaviors. This effort represents a move from developing the framework and mechanisms toward exploring the policy space.

2.2 Related Work

Attention to the issues of energy and power management is gaining momentum within both industry and academics.

Work by Flinn and Satyanarayanan on energy-aware adaptation using Odyssey [7] is closely related to our effort in several ways. Their fundamental technique differs in that it relies on the cooperation of applications to change the fidelity of data objects accessed in response to changes in resource availability. In contrast, our work focuses on managing global system resources in a unified manner. Unmodified applications and those that are not necessarily able to change "fidelity" benefit from our approach. Overall, we view our efforts as complementary: the operating system should manage global system devices in response to application requirements and the application should adapt its behavior when appropriate to reduce energy consumption.

Explicit energy management has also been designed for the Nemesis operating system [13]. This proposal describes how to extend the Nemesis resource accounting mechanisms, based on a calibration of device power consumption, to account for energy use by applications. Resource management is similar to Odyssey in that it is based on collaboration with applications. In Nemesis, this takes the form of an economic model for providing feedback to processes that allow them to adapt to shortages in energy availability.

Most of the literature on power/energy management has been dominated by consideration of individual components, in isolation, rather than taking a system-wide approach. Thus, there have been contributions addressing CPU frequency/voltage scheduling [6, 8, 14, 15, 20], disk spindown policies [3, 9, 11], memory page allocation [2, 12], and wireless networking protocols [10, 16]. The emphasis in much of this work has been on dynamically managing the range of power states offered by the devices. A recent paper [21] describes techniques involving buffer management policies and an API allowing application cooperation for shaping the disk request pattern to increase the effectiveness of disk spindown. This body of work is complementary to our currency model, as illustrated by our incorporation of spindown policies, and will impact the debiting policies for such devices in our

framework.

ECOSystem incorporates several ideas from previous work. The idea of currency borrows from the tickets abstraction of lottery scheduling [18, 19] with the value of a currency unit tied to energy. The key insight that distinguishes our work is that energy normalizes resource management across the diverse set of devices that consume it. We adopt the resource container abstraction [1] as our accounting entity in order to allow more complete accounting of activity and lazy receiver processing [4] in accounting for packet processing overhead.

3 Overview of Currency-based Policies

In our model, currency represents available global system resources. Currency allocation and accounting express and enforce policies to achieve energy-related goals. Next, we outline the various ways in which currency can be manipulated to implement a particular policy. The design space is rich, making an exhaustive exploration that fully utilizes all the mechanisms infeasible. However, Section 3.2 discusses several goals we want to achieve within the policy space. Section 3.3 introduces the applications and metrics used to evaluate our ability to achieve our goals.

3.1 Policy Building Blocks

1. Overall Currency Allocation The first decision point is the overall allocation of currency that determines how fast or how much energy can be consumed by the system as a whole. Choices include:

Per-epoch allocation level. We must determine the per-epoch currency availability based on the primary energy goal. Existing work focuses on achieving a target battery lifetime. Commonly used models of battery lifetime assume a constant power consumption, thus we impose a limit that translates directly into the currency allotment.

Epoch length. This determines the rate and granularity of currency allocation. Long epochs provide larger allocations and the ability to spend them in a more bursty fashion. Shorter epochs may smooth the consumption rate but pose problems accumulating enough for expensive operations. This issue is addressed in Section 7.

Dynamic adjustment. This concerns whether (and how) to allow dynamic adjustment of per-epoch allocation levels. One example is performing adjustments in allocation based on remaining capacity information from

a Smart Battery to correct for under-utilization of the resource (i.e., effectively a form of global redistribution of unused currency) or errors in the cost model.

2. Per-task Currency Allocations Given the overall allocation, the next decision is how to allocate currency among competing tasks.

Determination of per-task share. This may reflect an external priority or criticality of the task, the energy demand of the task, or some combination. In our prior work, the share is based on a user specification, scaled to a percentage based on all tasks in the system.

Handling of unused currency. When a task finishes an epoch without using its allocation, what happens to the residual currency? Choices include forfeiting the remaining allocation at the end of the epoch, saving it all, saving up to a dynamic or static *cap*, or distributing it among other tasks. Techniques to redistribute unused currency are considered in Section 4.

Debt limits. Do we allow a task to perform deficit spending and what are the rules on paying it back?

Subaccounts. Earmarking portions of a task's allowance for use with a particular device or by a particular thread within the resource container may require richer API support (a topic of future research).

3. Currency Accounting On the device side, various schemes may be appropriate for debiting tasks for access to devices. This may reflect actual energy costs or there may be rate structures designed to accomplish some energy objective. The strategies fall into the following categories:

Debiting. The straightforward policy is pay-as-you-go using the actual energy cost of the devices until currency is spent. In another scenario, prices levied against a task may dynamically vary to accomplish a subgoal (e.g., an extra "tax" to discourage use or a "sale price" to encourage use).

Bidding. The task may offer a price it is willing to support for access to an energy consuming resource. The bid does not necessarily imply that the task will be debited that amount for an activity.

Pricing. The price of a resource, which may be dynamically changing over time, is a way to encode thresholds in terms of currency and may interact with bids (e.g., in a negotiation protocol). Pricing may be decoupled from debiting to enforce threshold levels

without skewing accurate accounting for the resource. Pricing may also encode the power state of a device (e.g., the price of a disk access is discounted when the disk is already spinning and no spinup is required).

Examples of creative combinations of debiting, pricing, and bidding policies arise with the disk management policies in Section 8. We believe that expressing policies in terms of allocation and accounting operations on currency is a powerful way to unify resource management.

3.2 Currency-based Policies

The previous section has given an overview of the currency framework and the policy space that can be explored. In the introduction, we have articulated several energy-related goals that capture desirable behavior with the goal of achieving a target battery lifetime. In this section, we translate those goals more precisely in terms of our currency framework.

1. Reducing residual energy capacity. We have argued that, for certain applications, it is important to minimize residual energy capacity left when the target battery lifetime has been reached. Too much residual energy indicates an overly conservative management of the resource and lost opportunities for improved performance. We translate this into an allocation that is *currency conserving*. A currency conserving policy provides service in response to demand for energy as long as unspent currency is available in an epoch.

2. Proportional energy use. Ideally, the energy consumption of each task will match its assigned *share*. The energy consumption can be lower if the requirements of the task are low enough to be fully satisfied by the available level of energy. Even when currency allocations are appropriately adjusted to reflect demand, schedulers that gate access to devices may not offer opportunities to spend in proportion to allocations and may interfere with adaptations determining future allocations. We translate this goal of proportional energy use into device scheduling that is *aware of currency consumption/demand* throughout the system.

3. Coordination of multiple devices. Traditional resource management policies tend to concentrate on a single component of the system. For example, CPU scheduling algorithms are typically concerned only with tasks on the ready-to-run queue and allocation of CPU cycles. Processes blocked for device use have always posed subtle complications on CPU scheduling. With the focus on energy, the complications become more explicit since blocked processes can still be actively consuming energy. Tracking the consumption of currency

captures these interactions and allows the information to be incorporated into the scheduling policies of various devices in a coherent way.

4. Response time variation. The allocation of energy in epochs has the potential to cause large variations in response time and bursty behavior. One of our goals is to reduce the variation in response times. This translates into *carefully-paced* consumption of currency.

5. Energy efficiency. Encouraging the most efficient use of a device's power saving modes allows performance to be achieved at lower energy costs. This goal translates here into reducing the average currency cost per disk request by encouraging coalitions of tasks to share the overheads involved. Creative pricing strategies can reward such inter-task cooperation.

The challenge of unified global energy management is to explicitly address the kinds of interactions that are often hidden in per-device management.

3.3 Applications and Metrics for Evaluation

We intend to show that our currency model can be used to formulate policies to address the above goals. To evaluate our policies, we use several applications (described in Table 1) to create typical workload scenarios for a battery-constrained laptop user. We envision situations in which the user may want to have multiple tasks running concurrently (e.g., doing background jpeg encoding of a set of stored images while viewing the already encoded jpegs in slide show mode or listening to an MP3 while running through the slides of a PowerPoint presentation). For each experiment we use different combinations of these applications to emphasize specific aspects of the policy space. Each application presents a different set of demands for CPU, network bandwidth, disk I/O, or interactive "think time".

Within ECOSystem, we monitor the currency available for allocation each epoch and the currency consumed by each application during each epoch. It is also possible to track consumption by device. We then present our results in terms of average power (mW) derived from the amount of currency consumed or allocated per epoch. We also present appropriate application-specific performance metrics.

In the following five sections (Section 4 through Section 8), we illustrate the construction of currency-based policies to address each of the reformulated energy goals. Our goal in this paper is not to provide an optimal policy, but to show that policies formulated within the unified currency model offer desirable properties compared to more traditional (per-device) policies.

Application	Description	Demands
gqview	Image viewer	CPU, disk read, think time
ijpeg	SPEC2000 image encoding	CPU, image from disk or memory
RealPlayer	Video player	CPU, wireless, disk write
Netscape	Web browser	CPU, wireless, disk write, think time
x11amp	MP3 player	CPU, disk read
StarOffice	PPT presentation	CPU, disk read, think time

Table 1: Applications

4 Low Residual Energy Through Currency Conserving Allocation

Our first goal is to reduce residual energy. The details of our epoch-based currency allocation scheme are motivated by the overall goal of achieving a target battery lifetime by approximating a constant power consumption. To prevent large power peaks, our allocation policy caps the amount of unspent currency a task can save from epoch to epoch. Unspent currency that exceeds the cap is essentially thrown away, even if there is unmet demand by other tasks with insufficient currency. If there are enough instances of tasks that underspend their allocation during an epoch there can be a gradual accumulation of residual energy capacity because of the forfeited currency.

4.1 Currency Conserving Allocation

There are a number of ways to deal with the residual energy problem. One is to adjust the overall allocation level when the system detects that the battery is not being drained at the expected rate. If there is a consistent pattern of underspending by some tasks, the total allocation will grow, slowly at first, and be proportionally distributed to all tasks. Thus, needy tasks will benefit from receiving their share of a larger overall allocation.

Another approach is to explicitly redistribute excess currency to other tasks with insufficient currency for their energy demands. As a result of this approach, a task with a small energy share, determining its per-epoch allocation, may receive a large amount of excess currency. In this case, the task should have a large cap on its account balance to hold the extra currency. Similarly, for the tasks that consistently spend only a fraction of their energy share, the cap can be decreased to free the currency to the needy tasks. Specifically, we propose a two-step policy that first dynamically adjusts the per-task cap to reflect each task's energy needs (captured as the level of currency spent in previous epochs) as well as its specified share. The new cap is based on an exponential weighted average of currency expenditures in previous epochs. If the level of currency spent in the

most recent epoch is low relative to its history, then a lower weight factor (α_l) is used than otherwise (α_h). We have found that assigning weights that increase the cap quickly ($\alpha_h = 0.7$) and decrease it slowly ($\alpha_l = 0.1$) produces desirable behavior.

Second, the system redistributes currency amounts that overflow some task's cap to other tasks whose limits have not been reached. Specifically, our allocation algorithm behaves as follows: a) Every epoch, all containers receive currency proportionally unless their caps are reached. b) No currency is thrown away unless all containers reach their caps. c) Any currency overflow from a container is redistributed to other containers with unfilled capacity. For instance, for a total of n containers, if k (where $k < n$) containers reach their caps, the currency not allocated to the k containers is redistributed to the other $n - k$ containers, proportional to their shares. The first step of our allocation algorithm is to sort containers so that for any i (for $1 < i < n$), $container_i$ will not reach its cap unless $container_{i-1}$ reaches its cap first. The rest of the algorithm simply walks through the container list and does the following for each $container_i$, such that $0 < i < n$:

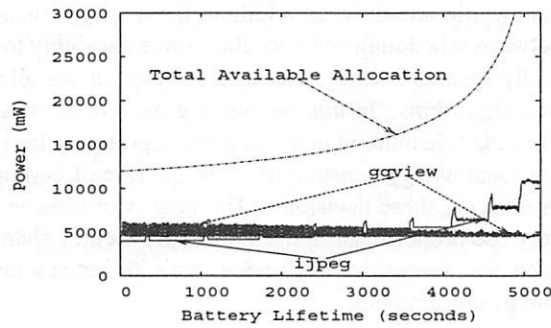
1. Calculate the entitled currency according to its energy share: $CurEntitled_i = \frac{CurAvail * CurShare_i}{\sum_{j=1}^n CurShare_j}$

where $CurAvail$ was initialized to be the total overall currency available in this epoch.

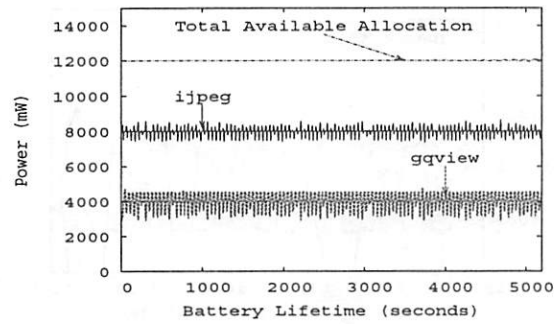
2. Calculate the allocated currency as the smaller of its entitled currency and the amount required to reach its cap: $CurAllocated_i = \min(CurCap_i, CurUnused_i, CurEntitled_i)$.

$CurUnused_i$ is the leftover currency in the container at the beginning of the epoch and $CurCap_i$ is the maximum amount of new currency can be added to the container.

3. Calculate the available currency for the rest of containers, gathering excess currency from the containers at the top of the sorted list: $CurAvail = CurAvail - CurAllocated_i$.



a) Original Allocation Policy



b) Currentcy Conserving Policy

Figure 1: Average Power Consumptions and Total Allocations for the Original and the Currentcy Conserving Allocation Schemes

The overall consumption should more closely match the overall allocation with this redistribution (at the risk of upsetting proportionality, considered in Section 5), thus reducing the residual energy. We refer to this algorithm as the Currentcy Conserving (CC) allocation policy.

4.2 Evaluation

As a qualitative argument, we note that without an explicit energy-related abstraction similar to currentcy, it is difficult to articulate precisely what residual energy means or identify means to enforce a target battery lifetime. Monitoring the state of the battery as a separate device-specific resource offers little in the way of control over the resource. Thus, there is no “traditional” baseline policy with which it makes sense to compare. We compare against the original currentcy allocation [22] with its battery-level feedback mechanism that adaptively adjusts overall allocation levels. In that original policy, residual energy accumulates if a task does not spend all of its currentcy and has exceeded its currentcy cap causing that unspent currentcy to be lost. We show that the original approach is less effective in reclaiming residual energy than explicit currentcy conservation.

To evaluate the benefits of currentcy conservation we use a workload consisting of the gqview image viewer and jpeg. Gqview is set to autobrowse mode where it continuously loads each of 12 images in a directory with a 10 second pause between each image. The images are copies of a high fidelity 0.5MB jpeg file, differing only in that each image has a unique number. The computationally intensive jpeg is run in a loop to continuously execute the SPEC command line, encoding and decoding an image from the reference data set residing in memory (SPEC command line options: -GO.findoptcomp vigo.ppm).

For this experiment, we set the target battery lifetime

at 90 minutes, and set desired shares of 66.6% for gqview and 33.3% for jpeg. These allocation settings correspond to an overall average power of 12000mW with 8000mW and 4000mW for gqview and jpeg, respectively. This represents an overly generous allocation to gqview which needs less than an average of 7000mW (c.f., Figure 3a). Jpeg, on the other hand, can easily consume up to 15.55W in the absence of other constraints.

Figure 1 shows our results. These plots show how the total allocation (presented in mW) changes over the lifetime of the battery. They also show the average power consumption of our two applications for the three managed devices. The per-epoch measurements have been smoothed using a centered moving average over a window of twenty-one data points.

From these data, we make several observations. First, for the original allocation policy (Figure 1a) we see that the total power available for allocation (the top curve) increases dramatically near the end of the target battery lifetime. There is approximately 6.7% of the original battery capacity remaining at the end. The simple redistribution approach that returns gqview’s unused currentcy (beyond the task’s cap) to the overall energy resource initially spreads the excess over a large number of epochs, but as the target battery lifetime approaches there is less time over which to spread the excess. Intuitively, each epoch consumes only a fraction of the total excess and thus available energy continues to grow. In addition, gqview still receives its share of the increasing overall allocation that it does not need.

The second observation we make based from Figure 1a is that as time progresses, gqview’s average power consumption (the middle line exhibiting some degree of scatter) decreases over time despite the increase in total availability. This is because the increase in available currentcy enables jpeg (the bottom solid line that steps up toward the end of the lifetime) to consume more and

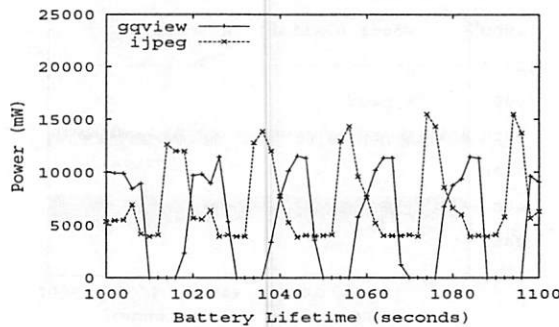


Figure 2: Time Varying Behavior with the Currency Conserving Allocation

more CPU time with the baseline CPU scheduling, undermining gqview's ability to execute when it needs to in order to consume its currency.

Figure 1b shows the average power consumption of gqview and jpeg when using the currency conserving allocation. We see that there is no significant change in the available allocation as we near the end of the target lifetime. Little residual energy capacity remains (less than 1%). By exploiting information in tasks' currency budgets, the currency conserving allocation policy successfully utilizes the available energy as compared to the approach that reacts to observed excess battery capacity. Formal analysis of these policies, using control systems theory, reveals that the original policy is unstable while the currency-conserving policy is stable.

There are variations in the power consumption of both applications due to gqview's execution variation that are not captured in the smoothed plots. To provide a better understanding of the simultaneous execution of jpeg and gqview, Figure 2 presents the power consumption in each epoch over a 100 second time interval. This figure shows that when gqview is idle (i.e., during "think" time with zero power consumption) jpeg can consume maximum CPU power. However, when gqview is active, jpeg is limited to its 4000mW allocation. There is a brief delay in this transition that occurs while jpeg's currency cap is adjusted. We observe that the power consumption of both gqview and jpeg can exceed their allocation share because of currency accumulating up to their caps.

5 Proportional Energy Use in CPU Scheduling

Even when allocations are appropriately proportional and consistent with actual demand, the ability to spend proportionally depends on policies that control access to resources, such as the schedulers for the CPU, network

bandwidth, and disk. In addition, there are interactions between scheduling and allocation since the ability to actually spend currency affects future caps in our allocation algorithms. In this section, we explore the role of the CPU scheduler in delivering the opportunity for proportional energy consumption and the role of currency in unifying these decisions. The goal is to achieve energy use proportional to the specified currency share of each task, unless the task's needs are satisfied at a lower energy consumption.

5.1 Energy Centric Scheduler

The base case for our explorations is the default Linux process scheduler, amended with the condition that the resource container of a process to be dispatched must contain available currency; otherwise, it is not considered ready to run again until the next epoch (when it receives a new infusion of currency). The amount of available currency in the task's *energy budget* is not a factor that influences the scheduling decision in any more substantial way than the ability to pay or not.

One might expect that by adapting a proportional scheduler to tasks' shares, better proportional energy use can be achieved. We consider stride scheduling [19] as representative of a local, CPU-only scheduler using each task's (static) share to determine its stride value.

Finally, we propose an energy-centric scheduler (EC scheduler) that accounts for the task's energy consumption (globally – regardless of where in the system the currency has been spent). The next process to be selected is one whose resource container has the lowest amount of currency spent relative to its specified share. This can be viewed as a bidding algorithm with the lowest bidder winning. As in traditional stride scheduling, an adjustment is made to "catch up" with the current pass when a process temporarily leaves the ready queue (e.g., blocked on synchronization or a synchronous I/O operation) and then rejoins.

To ensure that a process that is intermittently ready and blocked has sufficient opportunities to spend its currency, we can weigh the basis against which the energy consumed this epoch is compared by a factor defined to be the task's share divided by the amount of currency actually spent in the last epoch. This factor produces a *dynamic share* used to replace the fixed share value in the calculation of the task's stride. This biases allocation in favor of interactive tasks and helps them consume more of their share of currency whenever they are actively competing for the processor. This approach resembles compensation tickets from lottery scheduling and fractional quanta from stride scheduling [19], both of which give an advantage toward earlier scheduling of the next quantum to a task that voluntarily relinquished part of its

last quantum. The dynamic share is an adaptation that differs in two respects: it is based on a task's system-wide energy consumption and it applies over a longer period (spanning multiple quanta occurring during the current and previous epochs).

Our EC scheduler also incorporates one final feature called *self-pacing* (described in Section 7) with the goal of smoothing response times. Thus, there are three aspects of the EC scheduler that can be mixed in various combinations: the consumption-based stride, with or without dynamic shares, and with or without self-pacing. In this section, we consider the energy-centric scheduling with dynamic shares and without self-pacing.

5.2 Evaluation

Given a particular amount of currentcy per epoch, we investigate proportionally sharing this fixed allotment among competing tasks when some of the currentcy must be spent outside of the CPU. We analyze the effects of CPU scheduling using the default round-robin scheduler minimally modified to check for currentcy, the static energy-based stride scheduler, and our energy-centric scheduler with dynamic shares. We want to show that the energy-centric scheduler can achieve energy use that is proportional to the specified currentcy share of each task, allowing a lower consumption when it is enough to satisfy a task's performance needs. Thus, the first step is to determine whether there is a level of power consumption such that using more power does not produce significantly improved performance.

For the experiment presented, we simultaneously run gqview and jpeg with equal shares of a varying total allocation. First, we execute each application alone across the range of total allocation levels to see how the performance metric associated with that benchmark behaves. For gqview, configured with a think time of 10 seconds, the delay to completely display the given image decreases with increasing allocations of currentcy (mapped into average power for presentation) until around 6500mW where it levels out at approximately 6.3s. We pit gqview against jpeg, our CPU-bound benchmark that is always ready to run and whose performance metric, the delay to compress an image file, continues to decline until the maximum power consumption of the processor is reached (e.g., 15.55W). By setting the shares to be equal for the two competing applications, we are giving some benefit to the round robin and stride schedulers. In addition, the power needed by gqview for the disk (i.e., approximately 700mW) represents a relatively small level of consumption diverted to another device, making it more challenging for our energy-centric scheduler to distinguish itself.

Figure 3 shows our results. Figures 3a and 3c give the power consumed by gqview and jpeg as the allocation increases for each of the three scheduling policies. There are two additional lines on the plot for gqview showing the proportional allocation and the maximum power based on performance. Note that the bars representing the energy-centric scheduler show that gqview receives its appropriate energy share up until the point where it approaches its maximum power requirement. The Linux default scheduler and the energy-based stride scheduler both favor jpeg at the expense of gqview. In the case of the default Linux scheduler, this is because jpeg is always competing with gqview for the CPU and its round-robin algorithm gives each 50% and, for gqview, that is only when it is active (not during its think time or disk access). The static energy-based stride scheduler experiences similar problems when gqview and jpeg are competing for the CPU (with equal share values). Gqview is unnecessarily penalized for voluntarily reducing its energy consumption during idle periods.

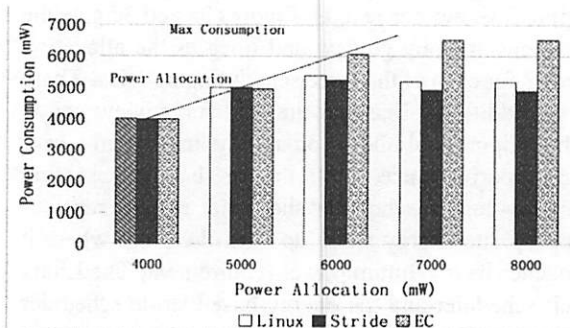
Our energy-centric scheduler extends the stride scheduler in two important ways. First, it selects the next task having the lowest amount of currentcy spent relative to its share, and second it dynamically computes a task's stride by including information about past consumption. This allows it to compensate for currentcy consumption of the other device as well as for periods of complete inactivity as in gqview's think time.

From Figures 3b and 3d, we see that with the energy-centric scheduler, gqview's delay approaches its performance of 6.3s when running without competition once it is given enough power. Neither the Linux scheduler or the stride scheduler deliver gqview that level of performance. Meanwhile, the performance level of jpeg is appropriate to its allocation level. It benefits from redistributed currentcy once gqview's consumption levels out, exceeding its expected performance (of running alone at that allocation level) for each scheduling choice.

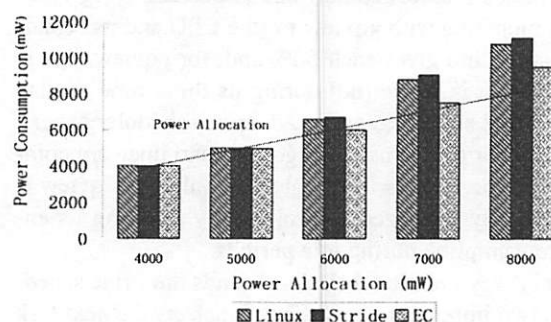
6 Coordinated Scheduling of Multiple Devices: Network Bandwidth and CPU

Currentcy is a unifying abstraction and proportional energy use extends to all other devices on a platform. Currently, ECOSystem only explicitly manages the CPU, NIC, and disk subsystem. The resource managers of various devices must cooperate toward a common goal such as proportional energy use. Otherwise, a bottleneck device with some other policy objective can disrupt currentcy flow in general.

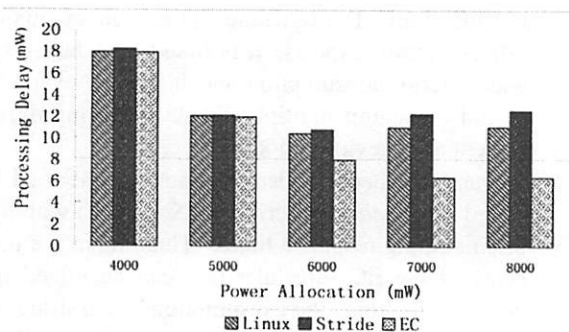
Given ECOSystem's currentcy model, tracking per-device consumption is straightforward for operations initiated via system calls. One particularly interesting chal-



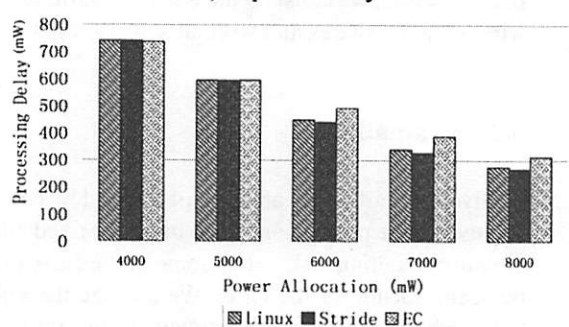
a) Gview Power Consumption



c) Ijpeg Power Consumption



b) Gview Delay



d) Ijpeg Delay

Figure 3: CPU Scheduling and Proportional Sharing of Energy

lenge to achieving proportional energy use is managing the wireless network bandwidth, especially for incoming packets. The tricky issue for incoming traffic is that by the time a packet has been received and management actions can be applied, the energy to receive it has already been consumed in the wireless card. This makes it difficult to selectively receive packets destined for tasks with available currency as opposed to tasks without currency.

We modified the Linux network packet processing code to implement a work conserving proportional bandwidth allocation policy. Our scheme identifies flows whose associated tasks have consumed bandwidth beyond their currency-determined share and reduces their allocated bandwidth. Assuming other tasks can consume released bandwidth, this bandwidth reduction continues until all connections consume bandwidth in proportion to their task's energy share. This is accomplished by explicitly reducing the advertised window to reduce a task's available bandwidth.

To create a stressful condition for evaluation where the network is the bottleneck, we set the wireless ethernet card to 1Mbps. We execute RealPlayer, Netscape, and jpeg with shares of 9000:3000:3000. Ijpeg serves as a CPU intensive application that does not compete for bandwidth but is capable of consuming 100% of the CPU. Realplayer plays a video clip rated at 550Kbs.

Netscape continuously reloads a web page with five images with zero think time. When executing without energy constraints, RealPlayer consumes about 10643mW to execute without pauses in video playback, while Netscape consumes 3115mW, running unconstrained. Both RealPlayer and Netscape can consume all of the network bandwidth available.

In all of our network experiments we use the currency conserving energy allocation policy and constrain the total power consumption to 15000mW. Since network bandwidth is the bottleneck resource for RealPlayer and Netscape and none of the applications' energy needs are satisfied, the goal is to achieve proportional overall energy use after satisfying the constraints of proportional network bandwidth and network energy consumption. Table 2 presents results for three of the scheduler design points: 1) Our energy-centric CPU scheduler with the default TCP implementation, 2) the default Linux CPU scheduler with an energy-centric network scheduler, and 3) our combined energy-centric CPU and network schedulers. We omit the case where neither the CPU or network scheduler are energy-aware.

Our results show that the conventional network scheduler fails to provide either proportional network bandwidth or energy consumption. This is because Netscape is allowed to consume an unfair portion of network bandwidth. Netscape is able to take more than 50% of the

Application	Allocation (mW)	CPU Power (mW)	Network Power (mW)	Disk Power (mW)	Total Power (mW)	Network Bandwidth (B/s)
Energy-Centric CPU Scheduler, Energy Oblivious Network						
RealPlayer	9000	2875	219	259	3354	3066
Netscape	3000	956	569	615	2142	7294
jpeg	3000	8960	23	0	8983	0
Default Linux CPU Scheduler, Energy-Centric Network						
RealPlayer	9000	5902	700	680	7282	8032
Netscape	3000	841	113	229	1182	2701
jpeg	3000	6611	18	0	6629	0
Energy-Centric CPU Scheduler, Energy-Centric Network						
RealPlayer	9000	8695	621	704	10020	8353
Netscape	3000	789	155	226	1170	2680
jpeg	3000	3778	10	0	3788	0

Table 2: Proportional Sharing: CPU and Network

bandwidth because it can open multiple connections. This reduces RealPlayer's ability to execute and produces an excess in currentcy that is reallocated to jpeg. This results in jpeg getting more of the CPU and consuming a much larger energy share than its intended allocation while the needs of the other applications are not satisfied (currentcy redistributed to jpeg would be considered acceptable if the other applications had their needs appropriately met).

When we use an energy-centric network scheduler, but the default Linux CPU scheduler, we see that bandwidth and network power are consumed by RealPlayer and Netscape closer to the specified ratio of 3 to 1. However, RealPlayer still suffers from competition for the CPU with jpeg which results in jpeg significantly exceeding its energy share.

The most satisfying results are obtained by using energy-centric schedulers for both the CPU and network. Both network bandwidth and network energy are consumed proportionally by RealPlayer and Netscape. RealPlayer's share of the CPU is also protected from jpeg by the energy-centric CPU scheduler. Netscape is throttled after receiving its share of network bandwidth (its bottleneck device) and can not consume the rest of its currentcy allocation. In this case, RealPlayer gets enough currentcy to meet its needs and execute without pausing.

7 Low Variance in Response Time Through Pacing Currentcy Expenditures

Given a case in which power consumption must be constrained, our epoch-based allocation has the potential

to produce bursty behavior if tasks consume currentcy as quickly as they can at the beginning of an epoch and then go idle after consuming their budget. One approach to smoothing consumption rates (and as a side-effect, response times), is to shorten the epoch.

Another approach to managing the rate of consumption is self-pacing in our EC scheduler. The idea is to delay a task if its consumption of currentcy is ahead of schedule during an epoch. Progress is defined as the amount of currentcy spent thus far in the current epoch divided by the task's budget for the epoch. If this progress is greater than the ratio of elapsed time in this epoch over epoch length, then the task is delayed and the processor may go idle for a short interval of time. This approach exploits the ability of currentcy to reflect an application's rate of progress. This approach to stretching execution is appropriate for a non-Dynamic Voltage Scaled processor. If available, DVS would be a preferred alternative to consider.

To compare these two approaches, we first look at the overhead of the first approach because it increases with the shortened epoch length and could become a performance bottleneck. However, experiments show the overhead for currentcy allocation is very small. Even if we perform allocation every 10ms (a timer interrupt occurs every 10ms, while the CPU scheduling quantum is 60ms in our system), the overhead is only 206 μ s for 18 resource containers and 40 μ s for 3 containers.

To explore the effects on response time of our two approaches for reducing bursty performance, we run Netscape and continuously load our department's web page. This page contains a banner image and some simple text. The autload is implemented using a javascript and this also allows us to measure the page load latency. This latency is composed of several http requests, dis-

Scheduling	Power Consumption (mW)				Delay (seconds)			
	CPU	Disk	Network	Total	Min	Max	Average	Std. Dev
Unthrottled	1047	1013	136	2197	0.27	0.68	0.43	0.11
Epoch	351	812	48	1212	1.0	33.8	3.8	5.8
Self-Pacing	313	842	43	1199	3.3	5.6	4.0	0.6

Table 3: Response Time Variation

playing the content and updating the disk cache. We set the think time between successive page loads at 2 seconds. Executing without any throttling requires about 2197mW.

We evaluate both an epoch-based approach that uses 0.01 second epochs, and the self-pacing approach with 10 second epochs. We allocate currency equivalent to an average of 1200mW to Netscape and measure 54 consecutive page loads for the self-paced test and 41 page loads for the epoch based approach. Differences are apparent in Table 3 when we examine the delay for a page load. Although the average delay is similar for the two policies, the self-paced scheduler has much lower variation in the delay. This can translate into a user perceived difference in performance as the self-pacing policy can provide a visibly smoother display of the web page. We note that similar visible differences occur when executing other applications, such as RealPlayer, Acrobat, and StarOffice.

8 Energy Efficient Disk I/O Through Cost-sharing

Encouraging more energy efficient use of devices is an important function of an energy centric operating system. Currency provides a means for passing along the savings to tasks that cooperate through their usage patterns. The disk presents unique challenges and opportunities for currency-based policies since it has non-uniform power consumption. The cost of spinning up the disk is much greater than keeping it spinning for a short duration. In this section, we consider techniques for more efficient disk access, focusing on sharing the spinup/spindown power costs. This introduces opportunities to work with debiting, bidding and pricing in the context of our currency model. The policy space for these approaches is very large, and many solutions may require an API for application involvement. For example, recent work [21] describes cooperative disk I/O operations that applications can use to facilitate such behavior. In this paper, we have limited our studies to techniques for managing disk access using pricing and bidding that can be implemented solely within the operating system without application involvement.

8.1 Shaping Access Patterns by Pricing and Bidding

Intuitively, we want to amortize spinups across multiple disk operations, which benefits from encouraging more bursty behavior. The key to more effectively manipulating the spinup/spindown behavior is *shaping the disk access patterns* to take advantage of this cost-sharing benefit within the debiting policy.

Pricing disk accesses can be used to reward a task for performing disk accesses in bursts. One approach we investigate sets the *entry price* of a disk access that requires a spinup cost much higher than the actual cost. When the access is actually permitted, we then debit the actual cost. This forces the task to accumulate enough currency to ensure that it can execute for a reasonable amount of time following the first access in hopes of generating more disk accesses while the disk is spinning.

We augment this pricing policy with the ability of tasks to bid on disk accesses. Tasks can indicate they are willing to contribute certain amounts toward the price of spinning up the disk. This is a natural place for API extensions. However, the OS can apply this technique transparently by checking the task's budget for sufficient surplus, analogous to a credit check. One goal of this technique is to enable multiple tasks to pool their currency and cooperatively use the disk in an energy-efficient manner.

Traditional techniques of skewing access patterns are amenable to currency-based variations. These include exploiting block caching and delaying writes while the disk is not spinning, piggybacking prefetching upon requests that spin up the disk on demand, and managing the buffer allocation. Thus, we explore a buffer allocation policy tied to the average disk access cost. Subject to limitations on the number of buffers systemwide, this policy attempts to reduce the costs (via effective prefetching, delayed writes) and make them uniform across tasks (which can tend to synchronize tasks into producing batches of disk activity).

We trigger prefetching operations and flushing of delayed writes that cause spinups using a bidding function based on the fraction of consumed buffers. Investigating the range of potentially useful bidding functions is clearly beyond the scope of this paper. We provide re-

sults for one bidding function that sets a bid offer to zero if less than 80% of the prefetch buffers are consumed otherwise to a weighted linear value ($bid = entry_price * (percent_buffers_consumed - 80) / (100 - 80)$). This corresponds to a function where value is greatly increased as the task nears a demand fetch. The disk flush daemon performs a large number of writes once it starts flushing pages to a spinning disk, writing back all dirty pages that have been idle for a more than 5 seconds. By contrast, the default Linux page flush policy is to check every 5 seconds for dirty pages that have not been accessed for 30 seconds and write those to disk.

8.2 Experiments on Disk Access Scheduling and Buffer Allocation

In this section, we show how the currentcy model enables policies based on pricing and bidding. First, we explore techniques to coschedule disk accesses for two applications with the goal of reducing overall disk power consumption. We present preliminary results on prefetching in our coordinated buffer management system.

Accesses In our first experiment we execute *jpeg* and *gqview* concurrently, and each application demand fetches data from the disk. *Jpeg* performs image compression on a set of ppm format image files. Each file is a copy of the same SPEC input (command line options: `-GO.compress vigo.ppm`) that is 2,359,355 bytes. When running unconstrained, *jpeg* requires about 2.452 seconds to process each file and start to read the next. *Gqview* displays the same set of image files using autobrowse mode with a 50 second think time. We set the total power allocation to 1500mW and the two tasks each get an equal share of 750mW. These severe constraints are used to accentuate the disk's impact on performance. The entry price for initiating a disk access is set to 24000mJ (twice the combined cost of spinup and spindown). We use an immediate disk spindown.

Without pricing/bidding, the total average disk power consumption is 911mW, with 403mW and 508mW for *jpeg* and *gqview*, respectively. Our currentcy-based policy formulated in terms of pricing/bidding reduces this value to 655mW (313mW *jpeg* and 342 *gqview*) by engineering more task cooperation in disk spinup sessions. Furthermore, the performance of each application improves, particularly *jpeg* which requires only 57 seconds to process the file compared to 74 without pricing/bidding.

The next experiment is designed to show the benefits of bidding for energy efficient disk prefetching. We set the total allocation at 1500mW and execute *jpeg* (same input as above) with 300mW concurrently with

the MP3 player, *x11amp*, which receives 1200mW allocation. *X11amp* reads a 3MB file, and is amenable to prefetching because of its sequential access pattern. We use the combined pricing/bidding approach where there is a high entry price for a disk spinup and *x11amp* contributes by bidding based on its prefetch buffer consumption. The average total disk power consumption is 357mW compared to 565mW without pricing/bidding. *Ijpeg*'s average disk power consumption reduces from 365mW to 229mW and its performance improves from 90 seconds per file to 66 seconds. *X11amp*'s disk power consumption reduces from 200mW to 128mW, and it does not incur any pauses in either policy.

Buffer Allocation It is also effective to balance the buffer allocation among prefetching-friendly tasks to facilitate more globally synchronized disk activity. To show the benefit of cooperation, we compare local and global prefetching behavior for two applications (*x11amp* and *StarOffice*) that exhibit sequential access patterns, since the unified buffer cache in Linux can easily detect these sequences and initiate prefetching. The spindown timeout is set to 1 second. *X11amp* reads a 3MB file, while *StarOffice* reads an 11MB presentation with 14 slides and executes in auto-transition mode with approximately 20 seconds between slides.

When prefetching is performed locally using the default Linux buffer allocation of 32 buffers for each task, the spinup and spindown costs are incurred for each task. The disk power consumption for *x11amp* is 186mW and *StarOffice* consumes 219mW for a combined 405mW.

In contrast, a global prefetching policy synchronizes the prefetching operations of the two tasks by allocating prefetch buffers according to a task's average disk access cost, which is determined by the task's buffer consumption rate. In this experiment, *x11amp* requires 256 prefetch buffers and *StarOffice* uses 1000. This significantly reduces the total disk power consumption to 280mW, with 65mW for *X11amp* and 215mW for *StarOffice*. *StarOffice* receives very little benefit since it is dominated by the cost to actually read the data, whereas *X11amp* leverages *StarOffice*'s relatively large number of disk accesses.

9 Conclusion

Energy management is an increasingly important aspect of system design. Our previously proposed currentcy model provides the framework for the operating system to manage energy as a first-class resource. This paper demonstrates that the currentcy model can be used to specify energy management policies that span multiple devices and diverse applications.

Using our ECOSystem prototype, we implement several currentcy-based policies, including: currentcy conserving scheduling algorithms that reduce residual battery capacity, proportional energy sharing, self-pacing to smooth response time variation, and energy efficient disk management. Our results show that the currentcy model is a powerful framework for expressing energy management policies and that our currentcy-based policies, by being able to capture aspects of global energy use, provide more coherency to system-wide energy management.

References

- [1] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Third Symposium on Operating Systems Design and Implementation*, February 1999.
- [2] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramiam, and M.J. Irwin. DRAM energy management using software and hardware directed power mode control. In *Proceedings of 7th Int'l Symposium on High Performance Computer Architecture*, January 2001.
- [3] Fred Douglass, P. Krishnan, and Brian Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *2nd USENIX Symposium on Mobile and Location-Independent Computing*, April 1995. Monterey CA.
- [4] Peter Druschel and Gaurav Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Operating Systems Design and Implementation*, October 1996.
- [5] C. S. Ellis. The Case for Higher-Level Power Management. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, March 1999.
- [6] Krisztian Flautner and Trevor Mudge. Vertigo: Automatic performance-setting for linux. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.
- [7] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles (SOSP)*, pages 48–63, December 1999.
- [8] D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.
- [9] D. Helmbold, D. Long, and B. Sherrod. A Dynamic Disk Spin-Down Technique for Mobile Computing. In *Proc. of the 2nd ACM International Conf. on Mobile Computing (MOBICOM96)*, pages 130–142, November 1996.
- [10] R. Kravets and P. Krishnan. Power Management Techniques for Mobile Communication. In *Proc. of the 4th International Conf. on Mobile Computing and Networking (MOBICOM98)*, pages 157–168, October 1998.
- [11] P. Krishnan, P. Long, and J. Vitter. Adaptive Disk Spin-Down via Optimal Rent-to-Buy in Probabilistic Environments. In *Proceedings of the 12th International Conference on Machine Learning*, pages 322–330, July 1995.
- [12] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla S. Ellis. Power aware page allocation. In *Proceedings of Ninth International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS IX)*, pages 105–116, November 2000.
- [13] Rolf Neugebauer and Derek McAuley. Energy is just another resource: Energy accounting and energy pricing in the Nemesis OS. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.
- [14] Trevor Pering, Thomas D. Burd, and Robert W. Brodersen. The Simulation and Evaluation of Dynamic Scaling Algorithms. In *Proceedings of the International Symposium on Low Power Electronics and Design*, August 1998.
- [15] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th symposium on operating systems principles*, pages 89–102, October 2001.
- [16] Mark Stemm and Randy Katz. Measuring and Reducing Energy Consumption of Network Interfaces in Hand-Held Devices. In *Proceedings of 3rd International Workshop on Mobile Multimedia Communications (MoMuC-3)*, September 1996.
- [17] Amin Vahdat, Carla Ellis, and Alvin Lebeck. Every Joule is Precious: The Case for Revisiting Operating System Design for Energy Efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.
- [18] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional share resource management. In *Proceedings of Symposium on Operating Systems Design and Implementation*, November 1994.
- [19] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical Report MIT/LCS/TR-667, MIT, 1995.
- [20] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for Reduced CPU Energy. In *USENIX Association, Proceedings of First Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994. Monterey CA.
- [21] Andreas Weissel, Bjorn Beutel, and Frank Bellosa. Co-operative I/O – a novel I/O semantics for energy-aware applications. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.
- [22] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, page 123, October 2002.

Design and Implementation of Power-Aware Virtual Memory*

Hai Huang, Padmanabhan Pillai, Kang G. Shin

Department of Electrical Engineering and Computer Science

The University of Michigan

Ann Arbor, MI 48109-2122

{haih,pillai,kgshin}@eecs.umich.edu

Abstract

Despite constant improvements in fabrication technology, hardware components are consuming more power than ever. With the ever-increasing demand for higher performance in highly-integrated systems, and as battery technology falls further behind, managing energy is becoming critically important to various embedded and mobile systems. In this paper, we propose and implement power-aware virtual memory to reduce the energy consumed by the memory in response to workloads becoming increasingly data-centric. We can use the power management features in current memory technology to put individual memory devices into low power modes dynamically under software control to reduce the power dissipation. However, it is imperative that any techniques employed weigh memory energy savings against any potential energy increases in other system components due to performance degradation of the memory. Using a novel power-aware virtual memory implementation, we estimate a significant reduction in memory power dissipation, from 4.1 W to 0.5–2.7 W, based on Rambus memory specifications, while running various real-world applications in a working Linux system. Unfortunately, due to a hardware bug in the chipset, direct power measurement is currently not possible. Applying more advanced techniques, we can reduce power dissipation further to 0.2–1.7 W, depending on the actual workload, with negligible effects on performance. We also show this work is applicable to other memory architectures, and is orthogonal to previously-proposed hardware-controlled power-management techniques, so it can be applied simultaneously to further enhance energy conservation in a variety of platforms.

1 Introduction

Limiting the energy consumption in mobile/embedded systems such as laptops, personal digital assistants (PDAs) and cellular phones is becoming increasingly important as they become widely used and accepted. With this large user community and a highly competitive market comes the inevitable demand for integrating more features and increased performance into small devices, which, in turn, comes at a cost of increased power dissipation. Products compete based on form factors (smaller and lighter is better) as well as additional features and

improved user experience, provided by fast processors, copious memory, resource-demanding software, and power-hungry hardware, thus making energy a precious resource. With hardware continuously improving in performance and price, vendors are able to build systems with higher-performance and higher-power components trying to meet users' ever-increasing demands and compete for customers. However, this results in systems that are over-provisioned with components that provide more capacity, more throughput, and more processing power than needed for the typical workload, and as a result, it is becoming more difficult to maintain long battery life in these devices. To make the situation worse, battery technology is improving at a much slower pace than hardware technology, making the gap between energy supply and demand increasingly larger. To deal with this emerging energy crisis, power management is becoming a more critical task than ever before.

Current hardware technologies allow various system components (e.g., microprocessor, memory, hard disk) to operate at different power levels and corresponding performance levels. Previous research has shown that by judiciously exploiting these power levels, depending on the workload, it is possible to have energy-limited systems built from high-performance and high-peak-power components with a minimal impact on the battery life of the device. The trick is to manage these power levels for each component intelligently based on the actual workload. For idle/normal workloads, some hardware components can be put at lower power levels or even be turned off. On the other hand, during peak workloads, the relevant hardware components are powered up to optimize for performance and provide responsive, high-quality service to users. Workloads on mobile systems such as laptops or PDAs are typically interactive, e.g., text editing, emailing, web surfing, or presenting PowerPoint slides, and due to the slow response time of human users, there are ample opportunities [13] to conserve energy by reducing power levels in various system components without any user-perceived performance degradation. During short intervals of high workload, e.g., switching slides, or re-computing a spreadsheet, the relevant system components can be briefly brought back to the higher-performance/power levels, effectively giving the user the benefits of both low-power

*The work reported in this paper was supported in part by the US Air Force Office of Scientific Research (AFOSR) under grant F49620-01-1-0120.

and high-performance in a single system. However, due to non-negligible transitioning delays of some hardware components, performance/energy may suffer if not handled properly.

A large body of previous research concentrates on reducing the power dissipation of microprocessors due to their high peak-power. Using existing techniques, a Mobile Pentium 4 processor dissipates only 1–2 W on average when running typical office applications despite having a high 30 W peak-power [17]. From a software perspective, further effort to reduce power in microprocessors is likely to yield only a diminishing marginal return. On the other hand, there has been relatively little work done on reducing power used by the memory. As applications are becoming more data-centric, more power is needed to sustain a higher-capacity/performance memory system. Unlike microprocessors, a fairly substantial amount of power is continuously dissipated by the memory in the background independent of the current workload. Therefore, the energy consumed by the memory is usually as much as, and sometimes more than, that of the microprocessor in a system. Implementing Power-Aware Virtual Memory (PAVM) allows us to significantly reduce power dissipated by the memory. In the rest of the paper, we describe our experiences in designing and implementing PAVM in a working system.

Our contributions in this paper are summarized as follows.

- Design of a PAVM system that reduces the overall power dissipation by minimizing the energy footprint of each process in a system.
- Exploration of techniques to reconfigure page allocations dynamically to yield additional energy savings by further reducing per-process energy footprint.
- Use of the Non-Uniform Memory Access (NUMA) techniques, in a novel way, as an abstraction layer to manage memory nodes in reducing power.
- Characterization of the memory usage pattern of processes in a Linux operating system, which allows PAVM to effectively manage all system memory including kernel memory, dynamically-loaded libraries, user process's own private pages, and their interactions.
- Implementation of PAVM in a real, working system (running Linux kernel 2.4.18) to evaluate the effectiveness of our techniques on various SDRAM architectures including SDR, DDR and RDRAM when running real-world applications.

The rest of the paper is organized as follows. Section 2 provides some background information on various memory technologies. Section 3 describes our initial design of PAVM, while Section 4 describes the limitations of this prototype design and the necessary modifications needed to handle the complexity of memory management and task interactions in a real, working implementation. Section 5 presents detailed experimentation results. In Section 6, we discuss the related

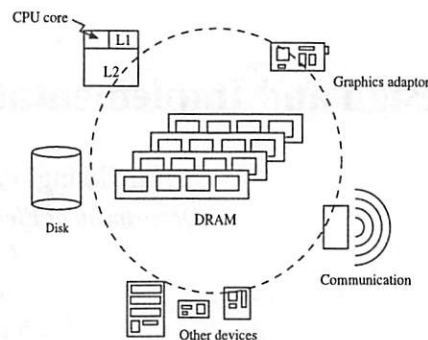


Figure 1: Interaction of memory with the rest of the system.

work. Finally, additional remarks about PAVM and conclusion are given in Sections 7 and 8, respectively.

2 Background

To understand how to reduce power in memory, we must first understand the current memory technologies and the interactions between memory and the rest of the system.

Since 1980, processor speeds have been improving at an annual rate of 80%, while Dynamic Random Access Memory (DRAM) only improved at an annual rate of 7% [39]. Even with a cache hierarchy sitting between the memory and the processor, hiding some of the latencies, the performance gap between the memory and the processor is continuously widening. Memory also interacts with various other system components, such as hard disks, video adapters, and communication devices that use DMA to transfer data as shown in Figure 1. Therefore, memory performance has a significant impact on the overall system performance. Since power reduction is only possible when the memory is operating at lower performance levels, it is critical to implement power-management techniques so that the power reduction in memory justifies any performance degradation, or even power increase, in other system components.

DRAM memory consists of large arrays of cells, each of which uses a transistor-capacitor pair to store a single bit as shown in Figure 2. To counter current leakage, each capacitor must be periodically refreshed to retain its bit information, making memory a continuous energy consumer. Because DRAM fabrication uses advanced process technologies that allow high-capacitance and low-leakage circuits, this refresh occurs relatively infrequently, and is not the largest consumer of DRAM power. Due to the large arrays with very long, highly-loaded internal bus lines, and high degree of parallel operations, significant energy is consumed by row decoders, column decoders, sense amplifiers, and external bus drivers. To reduce power, when a device is not actively being accessed, we can put it into lower power levels by disabling some or all of these subcomponents. However, when it is accessed again, a

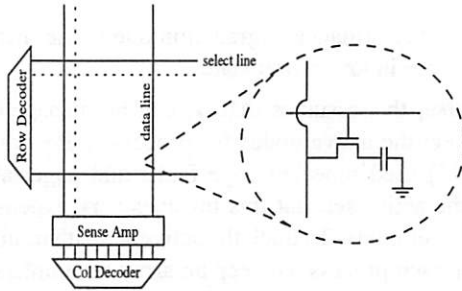


Figure 2: An overview of DRAM architecture with a magnified view of a single DRAM cell composed of a transistor-capacitor pair.

performance penalty is incurred for transitioning from a low-power mode to an active mode by re-enabling these components. This is due to the time needed to power up bus and sense amplifiers, and synchronize with the external clock, so this time penalty is called a *resynchronization cost*. This non-negligible resynchronization cost is the source of performance degradation when power management is not carefully implemented.

The above holds true for all Synchronous DRAM (SDRAM) architectures including the single-data-rate (SDR), the double-data-rate (DDR), and the recently-introduced Rambus (RDRAM) architectures. However, for our energy-conservation purposes, RDRAM differs from the rest by allowing a finer-grained unit of control in power management. In this paper, we consider all three memory types, and show that a finer-grained control can save a significant amount of additional energy over the coarser-grained traditional memory architectures. We now look more closely at these memory architectures with respect to power dissipation.

2.1 SDRAM Architectures

All three types of memory — SDR, DDR, and RDRAM — are physically organized as *modules*, composed of multiple *devices*, each of which is an individually-packaged integrated circuit. The traditional SDR and DDR architectures use wide (64 or 72-bits) data buses at relatively low clock rates (typically, 100 or 133 MHz), and require all devices on the same module to operate in parallel.

In comparison, Rambus DRAM technology [33] transfers data on a narrower 16-bit data channel, operating at twice the

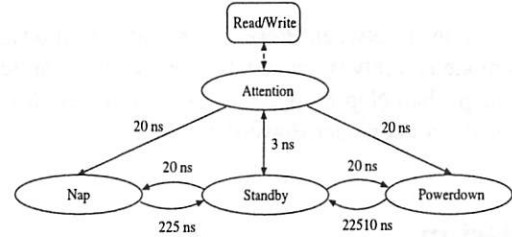


Figure 3: Possible power states for an RDRAM device. The transition time (in nanosecond) between two states is shown on the edge which connects them.

clock rate of 400 Mhz, to provide an extremely high throughput to better match the bandwidth needs of modern microprocessors. By using a narrow data bus, only a single device in the module needs to be actively transferring data at a time. This results in a lower power dissipation than for SDR or DDR, where all the devices in the module are activated in parallel to fill the wider bus. Furthermore, since they are not accessed in parallel, the devices in an RDRAM module can be in different power states, giving us a *device-granular* control in power management, in contrast to the traditional SDRAM architectures which can only provide *module-granular* control. Due to this finer-grained level of control, we will primarily focus on Rambus memory, although our approach is also applicable to SDR and DDR architectures, as we will show in Section 5.6.

There are four power levels of interest defined in the RDRAM specification, listed in decreasing order by power dissipation: Attention, Standby, Nap, and Powerdown. Devices are put into lower power levels by disabling the auxiliary subcomponents as discussed previously. For example, while in Attention mode, the self-refresh circuitry and the row/column decoders are active, and the internal clock is kept in-sync with the external clock generator, whereas in Powerdown mode, only the self-refresh circuitry is active to prevent data loss. The details of these power levels and the power dissipation of each are shown in Table 1. However, since read or write operations can only be performed on a memory device when it is in Attention mode, a resynchronization cost is incurred to return a device to Attention mode if it is in any lower power states. The possible state transitions and the corresponding resynchronization costs are shown in Figure 3.

Current RDRAM memory controllers already have a rudimentary form of power-saving policy built-in. Instead of having all devices in Attention mode, the memory controller puts all devices except for the one currently being accessed in Standby mode. Due to the small resynchronization time and the large power difference between Standby and Attention modes, power is significantly reduced with almost no performance loss. Using PAVM, we will show that an additional 59–94% power reduction can be achieved by exploiting the Nap mode with only a negligible performance overhead. Due to the large difference in resynchronization time and the small dif-

Power Level	Power	Active Components
Attention	313	Refresh, clock, row, col decoder
Standby	225	Refresh, clock, row decoder
Nap	11	Refresh, clock
Powerdown	7	Refresh

Table 1: Power dissipation (in mW) and active components used for a typical RDRAM device in various power levels.

ference in power between Powerdown and Nap modes, Powerdown mode is rarely more suitable for use in dynamic power management than Nap mode, as is verified in [9]. So, in this paper, we do not consider Powerdown mode.

3 Design

Prior research on reducing memory power dissipation mainly focuses on power management at a very low hardware level, where memory controllers are responsible for monitoring activity on each memory device and switching devices to lower power states based on various policies for detecting periods of inactivity. This has the benefit of being transparent to the running software, but as the controller is totally unaware of the processes that are using the memory on the system, performing power management at such a low level can often lead to poor decisions at a cost of decreased performance. In this paper, we elevate this decision making to the operating system level, where more information is readily available to make better transitioning decisions to minimize performance degradation and reap greater energy savings.

Before we delve into the design details of PAVM, we first introduce the concept of a memory *node*. We assume that the system memory is partitioned into one or more nodes, where a single node is the smallest unit of memory that can be power-managed independently of other memory. In SDR and DDR, therefore, a node corresponds to a memory module, which contains multiple memory devices, while for RDRAM, it corresponds to a single device within a module. This concept of a node generalizes the unit of control available for performing memory power management operations. We now describe how to manage the nodes to reduce power used by the memory.

3.1 Tracking Active Nodes

Since each node's power level can be separately controlled, total memory power can be reduced by selectively setting nodes to operate at lower power levels. However, selecting which nodes to put into lower power modes is critical to both system performance and power dissipation, since accessing a node in a low-power mode will incur resynchronization costs, stall execution, and, as a result, may increase energy consumption, offsetting any prior savings.

To avoid such costs, we need to ensure that all the nodes a process may access, i.e., its *active nodes*, are kept in high-power state. More specifically, we define a node to be an active node of process i if and only if at least one page from the node is mapped into process i 's address space, and we denote the set of active nodes for i as α_i . By promoting the nodes in α_i to Standby mode (high-power) and demoting all other nodes (i.e., those in $\bar{\alpha}_i$) to Nap mode (low-power) when process i is executing, we can reduce power while ensuring that process

i suffers no performance degradation due to the increased latency of nodes in low-power states.

Of course, this assumes that α_i can be managed to accurately reflect the active nodes for process i . Previous related research [4] used repeated page faults and page table scans to track the active set, but this involves very expensive, high overhead operations. To track the active set with minimal overheads, for each process we keep an array of counters, each of which is associated with a node in the system. The kernel is modified such that on all possible execution paths in which a page is allocated for, or mapped into, process i 's address space, the counter associated with the node containing this page is incremented. Similarly, when a page is unmapped, the counter is decremented. From these counters, α_i is trivially derived: a node is in α_i if and only if process i 's counter for the node is greater than zero. The overhead of maintaining α is only one extra instruction per mapping/unmapping operation, and is therefore negligible.

3.2 Reducing Active Set Size

Performing power management based on α 's, we can ensure that processes do not suffer any performance losses. However, this does not guarantee energy savings. In particular, if the size of the active set, $|\alpha|$, for each process is close to the total number of nodes in the system, power is not significantly reduced. So, to further reduce power dissipated by the memory, we need to minimize the total number of active nodes used per process for all processes in the system. This can be formally expressed as a minimization problem. Specifically, we want to minimize the summation, $(\sum \omega_i |\alpha_i| : i \in \text{all processes})$, where the number of active nodes, $|\alpha_i|$, for each process i is weighted by its CPU utilization (fraction of processing capacity/time spent executing the process), denoted by ω_i . Allocating pages for all processes among the nodes to minimize this sum is a difficult problem even with a static set of tasks, let alone in a dynamic system.

For simplicity, we assume that an approximate solution can be obtained by minimizing the number of active nodes for each process. To this end, a simple heuristic can be applied using the concept of a *preferred node* and maintaining a set of preferred nodes, ρ_i , for each process i . All processes start with an empty set ρ . When a process i allocates its first page, this page is taken from the node with the most free memory available, which is then added to ρ_i . Future memory allocations by this process are first tried on nodes in ρ_i . If all nodes in ρ_i are full, the allocation is again made from the node which currently has the most free memory available, and this new node is then added to ρ_i . By using this worst-fit algorithm to generate ρ , each process's memory footprint is packed into a small number of nodes, thereby decreasing each process's energy footprint.

3.3 A NUMA Management Layer

Implementing PAVM based on the above approaches is not easy on modern operating systems, where virtual memory (VM) is extensively used. Under the VM abstraction, all processes and most of the OS only need to be aware of their own virtual address spaces, and can be totally oblivious to the actual physical pages used. Effectively, the VM decouples page allocation requests from the underlying physical page allocator, hiding much of the complexities of memory management from the higher layers. Similarly, the decoupling of layers works in the other direction as well — the physical page allocator does not distinguish from which process a page request originates, and simply returns a random physical page, treating all memory uniformly. When performing power management on memory nodes, however, we cannot treat all memory as equivalent, since accessing a node in low-power state will incur increased latencies and overheads, and the physical memory address of allocated pages critically affects each process's energy footprint. Therefore, we need to eliminate this decoupling and make the page allocator conscious of the process requesting pages, so it can nonuniformly allocate pages based on ρ_i to minimize $|\alpha_i|$ for each process i .

This unequal treatment of sections of memory due to latencies and overheads for access is not limited to power-managed memory. Rather, it is a distinguishing characteristic of Non-Uniform Memory Access (NUMA) architectures, where there is a distinction between low-latency local memory and high-latency remote memory. In a traditional NUMA system, the notion of a *node* is more general than what we defined previously and can encompass a set of processors, memory pools, and I/O buses. The physical location of the pages used by a process is critical to its performance since intra- and inter-node memory access times can differ by a few orders of magnitude. Therefore, a strong emphasis has been placed on allocating and keeping the working set of a process localized to the local node.

In this work, by considering a node simply as a section of memory with a single common access time, for which the power mode can be set independently of other nodes, we can employ a NUMA management layer to simplify the nonuniform treatment of the physical memory. With a NUMA layer in place below the VM system, physical memory is partitioned into multiple nodes. Each node has a separate physical page allocator, to which page allocation requests are redirected by the NUMA layer. The VM is modified such that, when it requests a page on behalf of process i , it passes a hint (e.g., ρ_i) to the NUMA layer indicating the preferred node(s) from which the physical page should be allocated. If this optional hint is given, the NUMA layer simply invokes the physical page allocator that corresponds to the hinted node. If the allocation fails, ρ_i must be expanded as discussed previously. By using a NUMA layer, we can implement PAVM with preferential node allocation without having to re-implement complex low-level

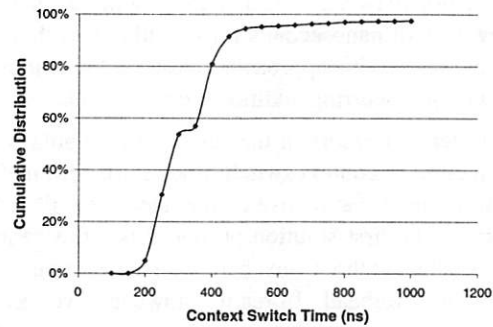


Figure 4: Cumulative distribution of context switch times.

physical page allocators.

3.4 Hiding Latency

Although the methods discussed so far ensure that a process experiences no performance loss during its execution, there still remains the issue of resynchronization latencies when transitioning power states of the nodes before a process is executed. As mentioned earlier, with RDRAM, switching a device from Nap to Standby mode requires 225 ns, which is not a very long time, but is nontrivial, as it would be incurred on every context switch. If this latency is not properly handled and hidden, it could, as a result of increased runtimes, erode the energy savings and undermine the techniques described above.

One possible solution is that at every scheduling point, we find not only the best process i to run, but also the second best process j . Before making a context switch to process i , we transition the union of the nodes in α_i and α_j to Standby mode. The idea here is that with a high probability, at the next scheduling point, we will either continue execution of process i or switch to process j . Effectively, the execution time of the current executing process will mask the resynchronization latency for the next process. Of course, the cost here is that more nodes need to be in Standby mode than needed for the current process, incurring greater energy costs, but, with a high probability, performance degrading latencies are eliminated.

A second solution is more elegant, has lower computational and energy overheads, and uses the context switching time to naturally mask resynchronization latencies. This is based on the fact that context switching takes time, due to loading new page tables and modifying internal kernel data structures, even before the next process's memory pages are touched. We instrumented the Linux 2.4.18 kernel to measure the portion of context switching time in the scheduler function after deciding which process to execute next, but before beginning its execution. The cumulative distribution of context switch times on a Pentium 4 processor clocked at 1.6 GHz is shown in Figure 4. From the figure, we can see that over 90% of all context switches take longer than 225 ns, and therefore, can fully mask the resynchronization latency for transitioning nodes from Nap to Standby mode. The sharp increase in the cumulative distri-

bution function between 175 and 225 ns indicates that we only pay a few tens of nanoseconds for the other less than 10% of context switches. This approach, therefore, hides most of the latency without incurring additional energy penalties.

With faster processors in the future, the cumulative distribution function of context switch times shifts left, making the second solution less attractive as the latencies will less likely be masked. The first solution proposed is more general and may be applied without any hardware constraints, but at a higher energy overhead. In reality, however, even as processor frequencies are rapidly increasing, context switch times improve rather slowly, so the second solution is viable under most circumstances.

4 Implementation

In this section, we describe our experiences in implementing and deploying PAVM in a real system. Due to complexities in real systems, a direct realization of the PAVM design described earlier does not perform up to our original expectations. Further investigation into how memory is used and managed in the Linux operating system reveals insights that we use to refine our original system and succeed in conserving a substantial amount of memory energy under the complex real-world environments.

4.1 Initial Implementation

Our first attempt to reduce memory power dissipation is a direct implementation of the PAVM design described in Section 3 within the Linux operating system. We extend the task structure to include the needed counters to keep track of the active node set, α_i , for each process i . As soon as the next-to-run process is determined in the scheduling function, but before switching contexts, the nodes in α of that process are transitioned to Standby mode, and the rest are transitioned to Nap mode. This way, power is reduced, the resynchronization time is masked by the context switch, and the process does not experience any performance loss.

We also modify page allocation to use the preferred set, ρ , to reduce the size of the active sets. Linux relies on the buddy system [20] to handle the underlying physical page allocations. Like most other page allocators, it treats all memory equally, and is only responsible for returning a free page if one is available, so the physical location of the returned page is generally nondeterministic. For our purpose, the physical location of the returned page is not only critical to the performance but also to the energy footprint of the requesting process. Instead of adding more complexity to an already-complicated buddy system, a NUMA management layer is placed between the buddy system and the VM, to handle the preferential treatment of nodes.

The NUMA management layer logically partitions all phys-

ical memory into multiple nodes and manages memory at a node granularity. The Linux kernel already has some node-specific data structures defined to accommodate architectures with NUMA support. To make the NUMA layer aware of the nodes in the system, we populate these structures with the node geometry, which includes the number of nodes in the system as well as the size of each node. As this information is needed before the physical page allocator (i.e., the buddy system) is instantiated, determining the node geometry is one of the first things we do at system initialization. On almost all architectures, node geometry can be obtained by probing a set of internal registers on the memory controller. On our testbed with 512 MB of RDRAM, we are able to correctly detect the 16 individual nodes, each consisting of a single 256 Mbit device. Node detection for other memory architectures can be done similarly.

Unfortunately, NUMA support for x86 in Linux is not complete. In particular, since the x86 architecture is strictly non-NUMA, some architecture-dependent kernel code was written with the underlying assumption of having only a single node. We remove these hard-coded assumptions and add multi-node support for x86. With this, page allocation is now a two-step process: (i) determine from which node to allocate, and (ii) do the actual allocation within that node. Node selection is implemented trivially by using a hint, passed from the VM layer, indicating the preferred node(s). If no hint is given, the behavior defaults to sequential allocation. The second step is handled simply by instantiating a separate buddy system on each node.

With the NUMA layer in place, the VM is modified such that with all page allocation requests, it passes ρ of the requesting process down to the NUMA layer as a hint. This ensures that allocations tend to localize in a minimal number of nodes for each process. In addition, on all possible execution paths, we ensure that the VM updates the appropriate counters to accurately bookkeep α and ρ for each process with minimal overheads, as discussed in Section 3.

4.2 Shared Memory Issues

Having debugged the new implementation, and ensured the system is stable with the new page allocation method, we evaluate PAVM's effectiveness at reducing energy footprints of processes. We expect that the active node set, α , for each task will tend to localize to the task's preferred node set, ρ . However, this is far from what we see.

Table 2 shows a partial snapshot of the processes in a running system, and, for each process i , indicates the nodes in sets ρ_i and α_i , as well as the number of pages allocated on each node.¹ It is clear from the snapshot that each process i has a large set of active nodes, where $|\alpha_i|$ is much larger than the corresponding $|\rho_i|$. This causes a significantly larger energy

¹We only show a partial list of processes running in the system due to space limitation, but other processes behave similarly.

Process	ρ	α								
<i>syslog</i>	14	0(3)	8(5)	9(51)	10(1)	11(1)	13(3)	14(76)		
<i>login</i>	11	0(12)	8(7)	9(112)	11(102)	12(5)	14(20)	15(1)		
<i>startx</i>	13	0(21)	7(12)	8(3)	9(7)	10(12)	11(25)	13(131)	14(43)	
<i>X</i>	12	0(125)	7(23)	8(47)	9(76)	10(223)	11(19)	12(1928)	13(82)	
			14(77)	15(182)						
<i>sawfish</i>	10	0(180)	7(5)	8(12)	9(1)	10(278)	13(25)	14(5)	15(233)	
<i>vim</i>	10,15	0(12)	9(218)	10(5322)	14(22)	15(4322)				
...

Table 2: A snapshot of processes’ node usage pattern using the initial version of PAVM. The number in parenthesis besides each active node indicates the number of pages the corresponding process is currently using on that node. Recall that our system has 16 nodes, denoted as 0, 1, ..., 15, each contains 256 Mbits (or 8192 4-KB pages).

footprint for each process than what we have originally anticipated. Nevertheless, since most pages are allocated in the preferred nodes, and none of the processes use all nodes in the system, we still consider this a working system that provides opportunities to put nodes into low-power modes and conserve energy. However, it is not as effective as we would like, due to the fact that for each process, there is a set of pages scattered across a large number of nodes.

To understand this “scattering” effect, we need to investigate how memory is used in the system. In most systems, a majority of the system memory is occupied by user processes. In turn, most of these pages are used to hold memory-mapped files, which include binary images of processes, dynamically-loaded libraries (DLL), as well as memory-mapped data files. To reduce the size of the executable binaries on disk and the processes’ cores in memory, DLLs are extensively used in Linux and most other modern operating systems. The scattering effect we observe is a result of the extensive use of DLLs combined with the behavior of the kernel-controlled page cache.

The page cache is used to buffer blocks previously read from the disk, so on subsequent accesses, they can be served without going to the disk, greatly reducing file access latencies. When a process requests a block that is already in the page cache, the kernel simply maps that page to the requesting process’s address space without allocating a new page. Since the block may have been previously requested by any arbitrary process, it can be on any arbitrary node, resulting in an increased memory footprint for the process. Unfortunately, this is not limited to shared data files, but also to DLLs, as these are basically treated as memory-mapped, read-only files. The pages used for DLLs are lazily loaded, through demand paging. So, when two processes with disjoint preferred nodes access the same library, the pages will scatter across the union of the two preferred node sets, depending on the access pattern of the processes and which process first incurred the page-fault to load a particular portion of the library into the page cache.

In the following sections, we describe the incremental changes we make to reduce the memory/energy footprint for each process by using DLL aggregation and page-migration techniques. We then discuss how to reduce overhead of these new techniques.

Process	ρ	α								
<i>syslog</i>	14	0(108)	1(2)	11(13)	14(17)					
<i>login</i>	11	0(148)	1(4)	11(98)	15(9)					
<i>startx</i>	13	0(217)	1(12)	13(25)						
<i>X</i>	12	0(125)	1(417)	9(76)	11(793)	12(928)	13(169)	14(15)		
<i>sawfish</i>	10	0(193)	1(281)	10(179)	13(25)	14(11)	15(50)			
<i>vim</i>	10,15	0(12)	1(240)	10(5322)	15(4322)					
...

Table 3: Effect of aggregating pages used by DLLs.

4.3 Revision #1: DLL Aggregation

Due to the many benefits of using dynamically-loaded libraries (e.g., *libc*), most, if not all processes make use of them, either explicitly or implicitly. Therefore, a substantial number of pages within each process’s address space may be shared through the use of DLLs. As discussed above, this sharing inevitably causes pages to be littered across memory, resulting in a drastic size-increase of α_i for each process i .

The cause of this scattering effect is that we are trying to load library pages into the preferred nodes of processes which initiated read-in from disk, as if these were private pages. To alleviate the scattering effect on the library pages, we need to treat them separately in the NUMA management layer. We implement this simply by ignoring the hint (ρ) that is passed down from the VM layer, and instead, resorting to a sequential first-touch policy, where we try to allocate pages linearly starting with node 0, and fill up each node before moving onto the next node. This ensures that all DLL pages are aggregated together, rather than scattered across a large number of nodes. Table 3 shows a snapshot of the same set of processes under the same workload as in Table 2, but with DLL aggregation employed.

As expected, aggregating DLL pages reduces the number of active nodes per process. However, a new problem is introduced. Due to the extensive use of DLLs, by grouping pages used for libraries onto the earlier nodes, we allocate a large number of pages onto these nodes and quickly fill them. As a result, processes need several of these low address nodes in their active sets to access all of the needed libraries. In the two snapshots shown, this is clearly apparent: after aggregation (Table 3), both nodes 0 and 1 are mapped in all of the process active sets, whereas only node 0 was needed without aggregation (Table 2). With many libraries loaded, we would use up these earlier nodes fairly quickly, and may increase the memory footprint of processes. We explain this in more details in the next section and also describe how to alleviate the extra burden on these earlier nodes.

4.4 Revision #2: Page Migration

Even after aggregating library pages, there is still some scattering of pages across nodes outside of ρ for each process. Some of this is due to actual sharing of pages, but the rest is due to previous sharing and residual effects of past file accesses in the page cache. Furthermore, even though aggregating all li-

Process	ρ	α
<i>syslog</i>	14	0(15) 14(125)
<i>login</i>	11	0(76) 11(183)
<i>startx</i>	13	0(172) 13(82)
<i>X</i>	12	0(225) 1(2) 12(2220)
<i>sawfish</i>	10	0(207) 1(56) 10(436)
<i>vim</i>	10,15	0(12) 1(240) 10(5322) 15(4322)
...

Table 4: Effect of library aggregation with page migration.

library pages ensures shared pages are kept in a few nodes, not all libraries are shared, or remain shared as the system execution progresses. It is better to keep these pages in the preferred nodes of the processes that are actively using them, rather than polluting nodes that are used for library aggregation and increasing the energy footprints of all processes. We can address all of these by using page migration.

In NUMA systems, page migration is used to keep the working set of a process local to the execution node in order to reduce average access latency and improve performance, particularly when the running processes are migrated to remote nodes for load-balancing purposes. In the context of PAVM, there is no concept of process migration, or remote and local nodes, but we can use the page-migration technique to localize the working set of a process to a fewer number of nodes and overcome the scattering effect of shared pages and items in the page cache. This will allow us to have more nodes in low-power states, thereby conserving more energy.

In our implementation, page migration is handled by a kernel thread called *kmigrated* running in the background. As with other Linux kernel threads, it wakes up periodically (every 3 seconds). Every time it wakes up, it first checks to see if the system is busy, and if so, it goes back to sleep to avoid causing performance degradation to the running processes. Otherwise, it scans the pages used by each process and starts migrating pages that meet certain conditions. We further limit any performance cost by setting a limit on the number of pages that may be migrated at each invocation of *kmigrated* to avoid spikes in memory traffic. Effectively, by avoiding performance overheads, we only pay a fixed energy cost for each page migrated.

A page is migrated if any of the following conditions holds.

- If a page is a process's *private* page (i.e., is used only by that process), and it is not on a node in that process's preferred set, ρ , then the page is migrated to any node in ρ . This will not affect the size of the active set, α , of other processes.
- If a page is *shared* between multiple processes, and the node that it resides on is outside of at least one of these processes' preferred sets (i.e., $\notin \bigcap \rho_i$), then the page is migrated to an earlier node so it can be aggregated with the other shared pages, if and only if this migration does not cause the size of α to increase for any of the processes sharing the page.

Migrating a process's private page is straightforward. We simply allocate a new page from any node in ρ of that process, copy the contents from the old page to the new page, redirect the corresponding page table entry in that process's page table to point to the new page, and finally free the old page.

Migrating a shared page is more difficult. First, from the physical address of the page alone, we need to quickly determine which processes are sharing this page so we can check if it meets the migration criterion given above. Second, after copying the page, we need a quick way to find the page table entry for each of the sharing processes, so we can remap the entries to point to the new page. If any of the above two conditions cannot be met, an expensive complete scan of the page tables of all processes is needed for migrating each shared page. Unfortunately, in the default Linux 2.4.18 kernel, neither requirement is met.

To our aid, Van Riel [34] has recently released the *rmap* kernel patch, a reverse mapping facility that meets both requirements nicely, and is included in the default kernel of the RedHat 7.3 Linux distribution. With *rmap*, if a page is used by at least one process, it will have a chain of back pointers (*pte_chain*) that indicates all page table entries among all processes that point to this page (meets the second requirement). In turn, for each page table containing the above page table entries, there is a back pointer indicating the process that uses this page mapping, satisfying the first requirement. So, when trying to migrate a shared page, we first allocate a new page, and find all the processes sharing this page to determine whether migrating this page will cause memory footprint to increase for any of the processes. If not, we copy the contents from the old page to the new page, replace all page entries that point to the old page with ones pointing to the new page, update the reverse mappings in the new page, and finally free the old page.

With *kmigrated* running, processes use much fewer nodes than in the initial version of the implementation, as shown in the snapshot in Table 4. In turn, memory power dissipation is significantly reduced for each process. However, for each page migrated, we incur a fixed energy cost for performing the memory-to-memory copies.

4.5 Revision #3: Reducing Migration Overhead

Although page migration greatly reduces the energy footprints of processes, it triggers additional memory activity, which may undermine the energy savings obtained. Thus, we must consider ways to limit the actual number of migrations to keep its benefits without incurring too much of energy cost. In this section, we propose two solutions to reduce the number of page migrations.

Solution 1: The DLL aggregation technique described pre-

Application	Interval	Light	Poweruser	Multimedia	Description
X+GNOME	continuous	x	x	x	runs X server using the default GNOME desktop environment
Mozilla	15 seconds	x	x		retrieves and displays webpages from randomly pre-generated URLs
XMMS	continuous	x	x		plays a stream of mp3 files
text editing	60 seconds	x	x		modifies a tex file, runs latex, bibtex, dvips, and displays it in ghostview
gcc	10 minutes		x		compiles Linux-2.4.18 kernel and kernel modules
Xine	continuous			x	plays an MPEG4-encoded movie in full-screen mode

Table 5: Description of the applications used in Light, Poweruser and Multimedia workloads.

viously assumes libraries tend to be shared. Any library that is not shared will later be migrated to the process preferred nodes. This is not efficient for those applications that use proprietary dynamic libraries. We can keep track of the processes that cause a large number of page migrations, and then classify them further as *private-page dominated* and *shared-page dominated*. A process is *private-page dominated* when the number of private pages migrated is much larger than the migrated shared pages. It indicates that the pages this process uses are less likely to be shared, meaning that we should allocate pages on this process's preferred node and not automatically aggregate the library pages it uses.

On the other hand, if a process is *shared-page dominated*, it means that many shared pages were wrongfully migrated initially and later migrated back. For these processes, we want to inhibit the number of page migrations for shared pages to prevent future migrations to correct the initial migration decision.

Solution 2: It is widely known that processes are short-lived. Process lifetime is similar to what is shown in Figure 5 [27], where only 2% of all processes live more than 30 seconds. Instead of performing page migration for all processes, we only migrate pages on behalf of long-lived processes, since the energy spent on migrating pages for short-lived processes does not justify the resulting energy savings. Note that the implementation of *kmigrated* implicitly avoids migrating all processes, as it checks the system at most once every 3 seconds, and only when the system is not busy, thus avoiding most short-lived processes.

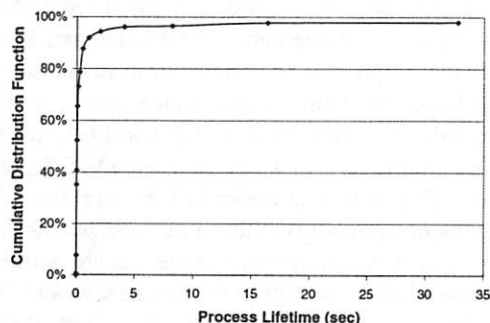


Figure 5: Cumulative distribution function of process life times.

5 Evaluation Results

Thus far, we have described how PAVM has been designed, implemented, and later evolved to improve energy efficiency of memory. In this section, we compare the effectiveness of the various power-management techniques implemented in PAVM with existing techniques to reduce memory's energy consumption. We first describe our experimental setup and test workloads, and then present extensive experimental results.

5.1 Experimental Workloads

Our goal in this section is to evaluate the effectiveness of PAVM when running real-world applications in a working system, and see how it compares to some other power-management policies. All workloads are executed on a Pentium 4 PC, with 512 MB of RDRAM (16 devices), running Linux 2.4.18 kernel. We define three types of workloads: Light, Poweruser, and Multimedia. These workloads are composed of different sets of user applications as shown in Table 5. The applications are not meant to be comprehensive, but rather found to be representative of the type of workloads used most often by us and our colleagues on mobile platforms.

Our representative Light workload consists of web browsing, with some e-mailing and word processing, while listening to mp3 music in the background, all run in a windowed, graphical environment. This type of workload is most commonly used on mobile platforms, and the workload is characterized as mostly idle. However, some users (Powerusers), due to the nature of their work (e.g., graphics designing, programming), utilize and stress their systems more. Their workloads can be characterized by repeated periods of low system utilization (designing, coding) followed by periods of high system utilization (rendering, compiling). To simulate this type of workload, we add periodic Linux kernel compilations to generate periods of heavy load on top of the Light workload. With a growing number of multimedia-rich applications, users may impose even heavier workloads on their mobile systems (e.g., 3D gaming, playing video). Multimedia workloads keep the system in a high utilization state continuously for a long period of time. To simulate this and keep workloads consistent across different experiments, we play an MPEG4-encoded movie using the Xine video player in full-screen mode.

T_E	Total elapsed time in the system
T_I	Total elapsed idle time in the system
$t_{i,j}^S$	Total time that node j operates in Standby mode while process i is active
$t_{i,j}^N$	Total time that node j operates in Nap mode while process i is active
f	Activity factor of memory transactions ²
U	Set of all processes in the system
P_A	Power dissipation of a node in Read/Write mode
P_S	Power dissipation of a node in Standby mode
P_N	Power dissipation of a node in Nap mode
V	Set of all nodes in the system

Table 6: Logged data and static parameters for calculating energy consumption.

5.2 Evaluation Methodology

Our PAVM implementation is currently fully operational, and has all the means to communicate with the RDRAM memory controller (i82850 chipset) on our Pentium 4 testbed to manage power by controlling the power state of individual nodes. However, due to a hardware bug found in the chipset [16], the system will hang when instructed to put a node in Nap mode. As a result, this prevents us from directly measuring the actual energy saved, e.g., with a digital power meter. However, by logging detailed information about the state of processes and the state of the system, combined with the information from memory device's datasheet, we can calculate fairly accurately how much energy would be consumed.

Specifically, to accurately calculate energy consumption, we need to log the operating times and memory use characteristics shown in the top portion of Table 6 from the running system. We also need some static system/memory parameters, shown in the bottom portion of Table 6, to complete the energy calculation.

Using these parameters, we can compute the energy consumed with the following equation:

$$\text{Energy} = |V|T_IP_i + \sum_{i \in U} \sum_{j \in V} (t_{i,j}^S P_S + t_{i,j}^N P_N) + f(T_E - T_I)(P_A - P_S). \quad (1)$$

where $P_i = P_N$ or P_S , depending on the power-management scheme used (see next section). This equation consists of three terms. The first is the energy consumed by the memory while the system is idle, and is simply the product of the number of nodes, total idle time, and either P_N or P_S , depending on whether the nodes are kept in Nap or Standby modes when system is idle. The second term computes the energy for keeping

²The activity factor, f , is obtained by dividing the number of memory transactions by the maximum possible number of memory transactions during non-idle time, $T_E - T_I$. The dividend is obtained from performance monitoring registers available on most modern processors, and the divisor is derived from the memory device's datasheet.

nodes in Nap and Standby modes while the system is not idle. This is a double summation over all processes and all nodes, where we weight the total time a particular process keeps a particular node in Nap and Standby modes by P_N and P_S , respectively. The last term reflects the additional energy required to actually read/write data from/to a memory device in Standby mode, and is a product of the total non-idle time, the additional power dissipated in Read/Write mode over Standby, and an activity factor, f , that gives the total number of memory transactions as a fraction of the maximum number possible when a device is kept in Read/Write mode (i.e., peak memory bandwidth).

5.3 Comparison of Basic Techniques

In this section, we compare three basic memory power-management techniques: the default built-in power-management policy implemented in current RDRAM memory controllers (*Base*), a simple *On/Off* technique, and our initial PAVM implementation. Recall from Section 2 that, under the *Base* policy, the controller keeps devices in Standby mode, and quickly switches them to Attention mode when accessed. The *On/Off* technique simply involves putting all nodes into Nap mode upon detecting system is idle, and restoring all nodes to Standby when any process is ready to run. It requires minimal kernel modifications to implement, and is worth considering here for its simplicity. PAVM, as described in Section 4.1, is compared with these two methods.

As the *Base* policy always keeps nodes in Standby, while the other two put all nodes into Nap mode when the system is idle, we use $P_i = P_S$ for *Base* and $P_i = P_N$ for the other policies when computing energy with Eq. 1. As neither *Base* nor *On/Off* uses Nap mode while processes are running, the second term in Eq. 1 simplifies to $|V|(T_E - T_I)P_S$ for these two policies.

To show how these power-management policies perform in real systems, we run the three workloads described earlier. The results are shown in Figures 6(a–c) for Light, Poweruser, and Multimedia workloads, respectively. Each graph shows cumulative energy consumed over time, normalized with respect to the *Base* policy. As one can see, for Light and Poweruser workloads, the simple *On/Off* policy performs well since it can exploit the large amount of idle time in the system to put nodes into Nap mode. With the Multimedia workload, idle time in the system is minuscule, and therefore, the *On/Off* policy approaches the *Base* policy at the end of the workload. PAVM, on the other hand, not only exploits idle time, but also reduces memory power dissipation when processes are actively running. Compared to the *On/Off* policy, it can save an additional 48–66%, 51–63%, 30–62% of energy for Light, Poweruser, and Multimedia workloads, respectively.

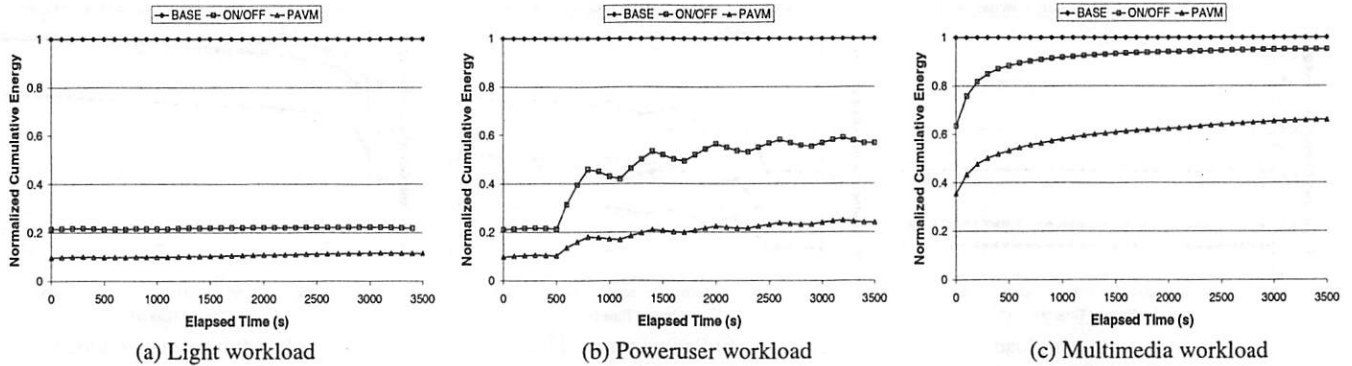


Figure 6: Cumulative energy for PAVM and On/Off policies, normalized to that of Base policy when running Light, Poweruser, and Multimedia workloads.

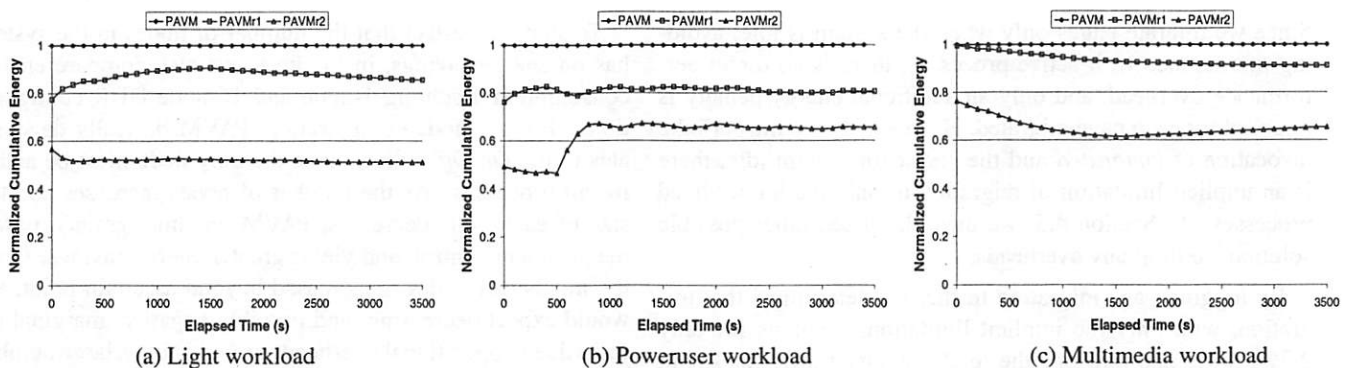


Figure 7: Cumulative energy for PAVM with library aggregation (PAVMr1) and PAVM with both library aggregation and page migration (PAVMr2), normalized to that of the initial PAVM implementation.

	Light	Poweruser	Multimedia
<i>Base</i>	4100 mW	4118 mW	4230 mW
<i>On/Off</i>	892 mW	2324 mW	3991 mW
<i>PAVM</i>	465 mW	986 mW	2687 mW
<i>PAVMr1</i>	397 mW	791 mW	2442 mW
<i>PAVMr2</i>	237 mW	646 mW	1725 mW

Table 7: Average memory power consumption for the different power-management policies, running various workloads over a one hour period.

5.4 Comparison of Advanced Techniques

Although the initial implementation of PAVM does very well compared to other basic techniques, we can conserve even more energy using more aggressive policies. In this section, we compare three versions of PAVM: the initial implementation of PAVM, revision 1 that uses library aggregation (PAVMr1), and revision 2 that also includes page migration (PAVMr2). Both of the aggressive policies try to keep nodes in the Nap mode longer, i.e., reducing $\sum t_{i,j}^S$ in the second term of Eq. (1), to realize significant additional energy savings.

We repeat the set of workloads under PAVMr1 and PAVMr2 policies, and the resulting energy consumption is plotted in Figures 7(a-c), normalized to that of the initial PAVM implementation. PAVMr1 saves an additional 0–20% and PAVMr2

saves an additional 25–50% of the energy relative to the initial implementation. It is interesting to note the jump at the 10-minute mark in the Poweruser workload for PAVMr2. This is the point at which the periodic Linux kernel compilation first runs. Since kernel compilation creates many short-lived processes that start and complete between invocations of *kmi-grated*, page migration does not help these processes, although it continues to be effective for the long-lived ones. Therefore, the benefit of page migration diminishes if short-lived processes dominate in the system.

The absolute average power dissipated for all of the power management techniques is summarized in Table 7. The *Base* system tends to draw close to a constant amount of power, since all nodes stay in Standby mode, with some small increases corresponding to the greater number of memory transactions in the Poweruser and Multimedia workloads. The energy savings realized vary greatly with the workload, and up to 94% reduction is seen with a lightly-loaded system. However, even with the very heavy Multimedia workload, 59% memory power reduction is realized.

5.5 Page-Migration Overhead

There is a significant energy improvement for PAVMr2 over PAVM, but this comes at a cost of page-migration overheads.

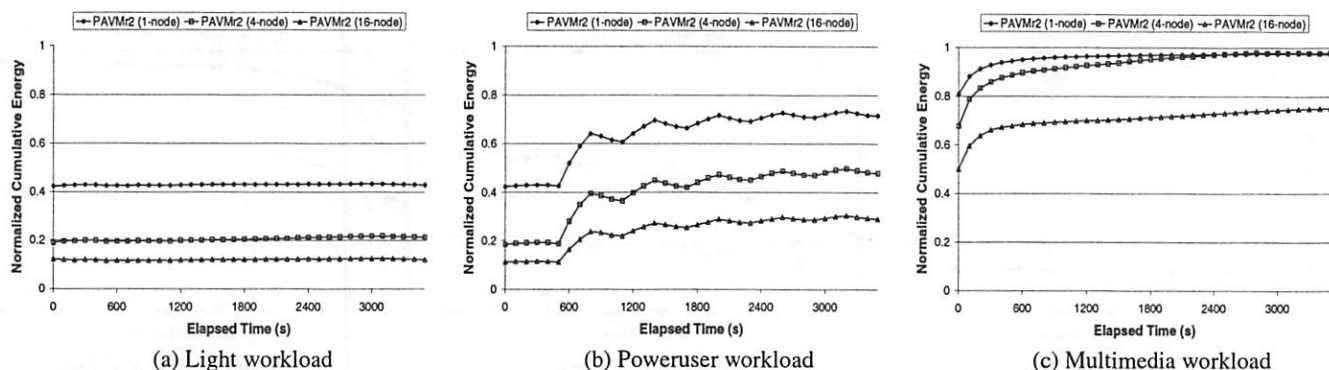


Figure 8: Effects of DDR's physical memory configuration on power dissipation under PAVMr2 when running Light, Poweruser and Multimedia workloads. Cumulative energy is normalized to the DDR Base policy.

Since we migrate pages only when the system is idle, avoiding interference with active processes, there is no direct performance overhead, and only an additional energy penalty is imposed for each page migrated. However, due to the periodic invocation of *kmigrated* and the check for system idle, there is an implicit limitation of migration to only the longer-lived processes. In Section 4.5, we have discussed other possible solutions to limit any overheads.

By logging page migration traffic, we determined that migration, with only the implicit limitation, accounts for only 2.7%, 0.8% and 0.8% of the total memory traffic for Light, Poweruser, and Multimedia workloads, respectively. If we adjust Figures 7(a-c) assuming maximal overhead reduction, we see no perceivable differences, so explicit attempts to reduce these overheads are not fruitful. Due to the significant energy savings and a fairly low overhead observed, page migration is beneficial in almost all circumstances.

5.6 Other Memory Architectures

We have discussed our power-management techniques primarily in the context of RDRAM architecture, but they are also applicable to other SDRAM architectures that support multiple operating power levels. In Figures 8(a-c), energy consumption is shown when running the same set of workloads discussed above, assuming a 4-node DDR memory configuration using PAVMr2. The cumulative energy is normalized against the default hardware-implemented policy for DDR. We still obtain significant energy savings, but not as much as with the previously-described RDRAM configuration.

There are two reasons for this. First, the power difference is much smaller between the DDR modes that correspond to RDRAM's Standby and Nap modes. Therefore, for DDR, putting nodes in "Nap" mode shows a smaller relative energy savings. Second, and more importantly, the notion of a node is coarser-grained for DDR than RDRAM. As discussed earlier, power management for DDR can only be done at the module-level, whereas in RDRAM, power can be adjusted at a device-level granularity, resulting in a much larger number of nodes.

To show the effect that the number of nodes in the system has on energy savings, in Figure 8, we also compare energy consumption assuming 1-node and 16-node DDR configurations. For a 1-node configuration, PAVM basically degenerates to the *On/Off* policy, since the only node must be active for all processes. As the number of nodes increases and the size of each node decreases, PAVM has finer-grained power management control, and yields greater energy savings. Once the number of nodes is increased beyond a certain point, we would expect decreasing, and possibly negative, marginal returns due to operational overheads of managing a large number of nodes. Finding the sweet spot that provides the maximum energy savings is system-/memory- dependent and beyond the scope of this paper. However, we believe that the 8- to 16-node granularity provided in most RDRAM configurations is not far from this sweet spot for typical mobile workloads. Furthermore, assuming that 4 nodes are available in a DDR system is probably optimistic, since in real systems, we are more likely to see 1-node and 2-node configurations, especially on mobile platforms. The results for SDR is similar to DDR, and due to the space limitation, are not shown here.

6 Related Work

Conserving energy in mobile and embedded systems is becoming an active area of research as hardware components are becoming more power-hungry than ever, and as battery technology is not able keep up with the growing demands. By exploiting the ability of modern hardware components to operate at multiple power levels, recent research has demonstrated that a significant amount of energy can be conserved. Due to the high-peak power demands of the processor, a large body of work has focused on reducing processor energy consumption. Weiser *et al.* [38] first demonstrated the effectiveness of using Dynamic Voltage Scaling (DVS) to reduce power dissipation in processors. Later work [2, 11, 14, 15, 25, 29–32] further explored the effectiveness of DVS techniques in both real-time and general-purpose systems.

There is also a large body of work that focused on reducing

power in other system components, including wireless communication [10, 18, 21, 36], disk drives [6, 7, 22, 24], flash [5, 28], cache [1, 19, 37], and main memory [3, 4, 8, 9, 23], while others [12, 26, 35, 40] explored system-level approaches to extend/target the battery lifetime of systems, as opposed to saving energy for individual components.

Among the works dealing with main memory energy, in [8, 23], Lebeck *et al.* studied the effects of various static and dynamic memory-controller policies to reduce power dissipated by the memory using extensive simulations. However, they assumed having additional hardware support to do very fine-grained idle time detection for each device so the controller can correlate this idle time with a power state for each device. In a later work, they used a stochastic Petri Nets approach to explore more complex policies [9]. Our work differs significantly in not assuming any additional hardware support or a particular memory architecture. Moreover, by elevating the decision-making to the OS level, we can use information known to the OS to conserve more energy without degrading performance. Finally, we have fully implemented a power-aware VM system that handles the complexities of a real, working system, and demonstrated its effectiveness when running real-world applications.

Delaluz *et al.* [3] took a *compiler-directed* approach, where power-state transition instructions are automatically inserted into compiled code based on offline profiling. The major drawback of this approach is that the compiler only works with one program at a time and has no information about other processes that may be present at runtime. Therefore, it needs to be either less aggressive or else it can trigger large performance and energy overheads when used in a multitasking system. This approach, however, is appropriate for DSP-like platforms where single-application systems are common.

Delaluz *et al.* [4] later showed a simple scheduler-based power-management policy. The basic idea is similar to our work, but is of much more limited scope. In our work, much effort is put into making the underlying physical page allocator to allocate pages by collaborating with the VM through a NUMA management layer so the energy footprint is reduced for each process, whereas they rely on the default page allocation and VM behaviors. As we have seen in Section 4.2, a substantial amount of power-saving opportunities remain unexploited even with our rudimentary implementation of PAVM, let alone when randomly allocating pages using the default page allocator. In [23], it was also noted that the default page allocation behavior has a detrimental impact on the energy footprints of processes. Second, we have explored advanced techniques such as library aggregation and page migration which are necessary for reducing memory footprints when complex sharing between processes in real operating systems is involved. Finally, in their work, the active nodes are determined using page faults and repeated scans of process page tables. Although this ensures only the truly active nodes are detected, it is intrusive and involves high operational

overheads. In contrast, we take every precaution to avoid performance overheads and hide any unavoidable latencies in our implementation, and the end result is a PAVM system that can save a significant amount of energy with only a very small performance overhead.

7 Discussion

In the current implementation, there are two limitations that we do not fully address. First, we do not consider direct memory access (DMA) by other hardware components on nodes that may be in reduced power states, which may result in performance degradation. This can be mitigated by ensuring that DMA uses only pages within a pre-defined physical memory range (e.g., the first node), which, due to the use of library aggregation, is almost always in Standby mode.³

Second, kernel threads that run in the background may touch random pages belonging to any process in the system. Since these maintenance threads are invoked fairly infrequently, a simple solution is to treat these as special processes and turn on all nodes when they are invoked to avoid performance degradation.

8 Conclusion and Future Work

Due to better processing technology and a highly competitive market, systems are equipped with bigger-capacity and higher-performance main memory as workloads are becoming more data-centric. As a result, power dissipated by the memory is becoming increasingly significant. In this paper, we have presented the design and analysis of power-aware virtual memory (PAVM) to reduce total memory energy expenditure by managing power states of individual memory nodes. We have also shown a working implementation of PAVM in the Linux kernel, and described how it was later evolved to handle complex memory sharing among multiple processes and between processes and the kernel in a modern operating system.

By performing extensive experiments with real applications, we are able to show that even with a rudimentary version of PAVM, we can save 34–89% of the energy normally consumed in a 16-device RDRAM memory configuration. By applying more advanced techniques such as DLL aggregation and page migration in PAVM, we are able to reduce energy dissipation by an additional 25–50%. We have also shown the applicability of this approach for other SDRAM architectures such as DDR and SDR, which can also benefit greatly under PAVM.

We have used a NUMA abstraction to organize and manage memory in our PAVM implementation, and have borrowed some NUMA concepts such as the notion of a node

³Note that due to limitations in older ISA hardware, Linux for x86 already has support to limit DMA transactions to the first 16 MB of memory (i.e., within the first node).

and the page-migration technique. In the future, we would like to explore if other NUMA techniques, such as page replication, can be effective in the context of energy conservation. In addition, we would also like to investigate the interactions between OS-controlled and hardware-implemented power-management policies to further decrease energy consumption of memory.

References

- [1] R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *International Symposium on Low Power Electronic Design*, 1998.
- [2] T. D. Burd and R. W. Brodersen. Energy efficient CMOS microprocessor design. In Trevor N. Mudge and Bruce D. Shriver, editors, *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 288–297. IEEE Computer Society Press, 1995.
- [3] V. Delaluz and *et al.* Dram energy management using software and hardware directed power mode control. In *International Symposium on High-Performance Computer Architecture*, 2001.
- [4] V. Delaluz and *et al.* Scheduler-based dram energy power management. In *Design Automation Conference* 39, 2002.
- [5] Fred Dougliis, Ramon Caceres, M. Frans Kaashoek, Kai Li, Brian Marsh, and Joshua A. Tauber. Storage alternatives for mobile computers. In *Operating Systems Design and Implementation*, pages 25–37, 1994.
- [6] Fred Dougliis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *USENIX Winter*, pages 292–306, 1994.
- [7] Fred Dougliis, Padmanabhan Krishnan, and Brian Bershad. Adaptive disk spin-down policies for mobile computers. In *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*, 1995.
- [8] X. Fan, C. S. Ellis, and A. R. Lebeck. Memory controller policies for dram power management. In *International Symposium on Low Power Electronics and Design*, 2001.
- [9] X. Fan, C. S. Ellis, and A. R. Lebeck. Modeling of dram power control policies using deterministic and stochastic petri nets. In *Workshop on Power-Aware Computer Systems*, 2002.
- [10] Laura Marie Feeney and Martin Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *IEEE INFOCOM*, 2001.
- [11] Krisztian Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01*, 2001.
- [12] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles*, pages 48–63, 1999.
- [13] Richard A. Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. In *USENIX Winter*, pages 201–212, 1995.
- [14] K. Govil, E. Chan, and H. Wassermann. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st Conference on Mobile Computing and Networking MOBICOM'95*, 1995.
- [15] Flavius Gruian. Hard real-time scheduling for low energy using stochastic data and DVS processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*, 2001.
- [16] Intel. <http://www.intel.com/design/chipsets/specupdt/>.
- [17] Intel. <http://www.intel.com/design/mobile/perfbref/250725.htm>.
- [18] Christine E. Jones, Krishna M. Sivalingam, Prathima Agrawal, and Jyh-Cheng Chen. A survey of energy efficient network protocols for wireless networks. *Wireless Networks*, 7(4):343–358, 2001.
- [19] M. Kamble and K. Ghose. Energy-efficiency of vlsi caches: A comparative study. In *Proc. of International Conference on VLSI Design*, 1997.
- [20] Donald E. Knuth. The art of computer programming. volume 1, pages 435–455, 1968.
- [21] Robin Kravets and P. Krishnan. Power management techniques for mobile communications. In *Proceedings of the 4th Conference on Mobile Computing and Networking MOBICOM'98*, 1998.
- [22] P. Krishnan, P. Long, and J. Vitter. Adaptive disk spin-down via optimal rent-to-buy in probabilistic environments. In *Proc. of International Conference on Machine Learning*, pages 322–330, 1995.
- [23] Alvin R. Lebeck and *et al.* Power aware page allocation. In *Architectural Support for Programming Languages and Operating Systems*, pages 105–116, 2000.
- [24] Kester Li, Roger Kumpf, Paul Horton, and Thomas E. Anderson. A quantitative analysis of disk drive power management in portable computers. In *USENIX Winter*, pages 279–291, 1994.
- [25] Jacob Lorch and Alan J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 50–61, 2001.
- [26] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Operating-system directed power reduction. In *International Symposium on Low Power Electronics and Design*, pages 37–42, 2000.
- [27] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Power-aware operating systems for interactive systems, 2002.
- [28] B. Marsh, F. Dougliis, and P. Krishnan. Flash memory file caching for mobile computers. In *To appear in Proceedings of the 27th Hawaii Conference on Systems Science*, 1994.
- [29] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low-Power*, 2000.
- [30] Trevor Pering, Tom Burd, and R. Brodersen. Voltage scheduling in the IpARM microprocessor system. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'00*, 2000.
- [31] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th ACM Symposium on Operating Systems Principles*, 2001.
- [32] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01*, 2001.
- [33] Rambus. http://www.rambus.com/technology/quickfind_documents.shtml#datasheets.
- [34] Rik V. Riel. <http://www.surreal.com>.
- [35] Tajana Simunic, Luca Benini, Peter Glynn, and Giovanni De Micheli. Dynamic power management for portable systems. In *International Conference on Mobile Computing and Networking*, pages 11–19, 2000.
- [36] M. Sterrm and R. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Communications*, vol.E80-B, no.8, p. 1125–31, E80-B(8):1125–31, 1997.
- [37] C. Su and A. Despain. Cache design tradeoffs for power and performance optimization: A case study. In *Proc. of the International Symposium on Low Power Design*, pages 63–68, 1995.
- [38] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, 1994.
- [39] W. Wulf and Sally McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.
- [40] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

Operating System Support for Virtual Machines

Samuel T. King, George W. Dunlap, Peter M. Chen

*Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
<http://www.eecs.umich.edu/CoVirt>*

Abstract: A virtual-machine monitor (VMM) is a useful technique for adding functionality below existing operating system and application software. One class of VMMs (called Type II VMMs) builds on the abstractions provided by a host operating system. Type II VMMs are elegant and convenient, but their performance is currently an order of magnitude slower than that achieved when running outside a virtual machine (a standalone system). In this paper, we examine the reasons for this large overhead for Type II VMMs. We find that a few simple extensions to a host operating system can make it a much faster platform for running a VMM. Taking advantage of these extensions reduces virtualization overhead for a Type II VMM to 14-35% overhead, even for workloads that exercise the virtual machine intensively.

1. Introduction

A virtual-machine monitor (VMM) is a layer of software that emulates the hardware of a complete computer system (Figure 1). The abstraction created by the

VMM is called a virtual machine. The hardware emulated by the VMM typically is similar or identical to the hardware on which the VMM is running.

Virtual machines were first developed and used in the 1960s, with the best-known example being IBM's VM/370 [Goldberg74]. Several properties of virtual machines have made them helpful for a wide variety of uses. First, they can create the illusion of multiple virtual machines on a single physical machine. These multiple virtual machines can be used to run applications on different operating systems, to allow students to experiment conveniently with building their own operating system [Nieh00], to enable existing operating systems to run on shared-memory multiprocessors [Bugnion97], and to simulate a network of independent computers. Second, virtual machines can provide a software environment for debugging operating systems that is more convenient than using a physical machine. Third, virtual machines provide a convenient interface for adding functionality, such as fault injection [Buchacker01], primary-backup replication [Bressoud96], and undoable disks. Finally, a VMM provides strong isolation

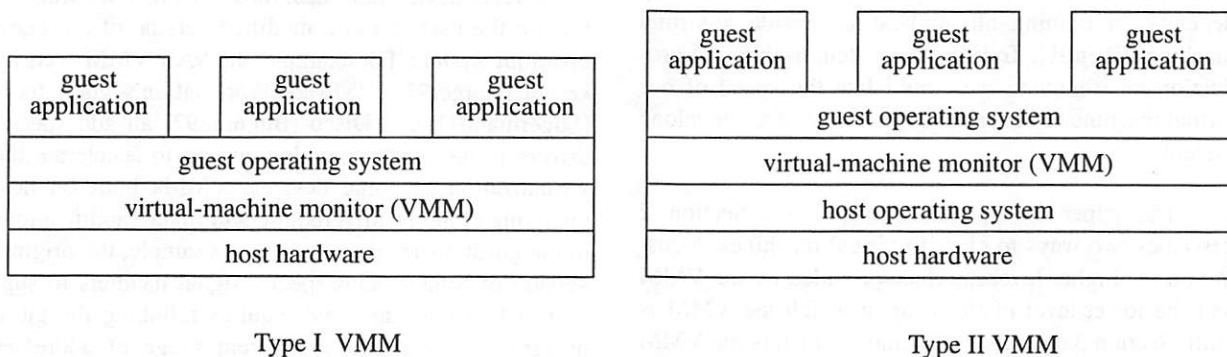


Figure 1: Virtual-machine structures. A virtual-machine monitor is a software layer that runs on a host platform and provides an abstraction of a complete computer system to higher-level software. The host platform may be the bare hardware (Type I VMM) or a host operating system (Type II VMM). The software running above the virtual-machine abstraction is called guest software (operating system and applications).

between virtual-machine instances. This isolation allows a single server to run multiple, untrusted applications safely [Whitaker02, Meushaw00] and to provide security services such as monitoring systems for intrusions [Chen01, Dunlap02, Barnett02].

As a layer of software, VMMs build on a lower-level hardware or software platform and provide an interface to higher-level software (Figure 1). In this paper, we are concerned with the lower-level platform that supports the VMM. This platform may be the bare hardware, or it may be a host operating system. Building the VMM directly on the hardware lowers overhead by reducing the number of software layers and enabling the VMM to take full advantage of the hardware capabilities. On the other hand, building the VMM on a host operating system simplifies the VMM by allowing it to use the host operating system's abstractions.

Our goal for this paper is to examine and reduce the performance overhead associated with running a VMM on a host operating system. Building it on a standard Linux host operating system leads to an order of magnitude performance degradation compared to running outside a virtual machine (a *standalone system*). However, we find that a few simple extensions to the host operating system reduces virtualization overhead to 14-35% overhead, which is comparable to the speed of virtual machines that run directly on the hardware.

The speed of a virtual machine plays a large part in determining the domains for which virtual machines can be used. Using virtual machines for debugging, student projects, and fault-injection experiments can be done even if virtualization overhead is quite high (e.g. 10x slowdown). However, using virtual machine in production environments requires virtualization overhead to be much lower. Our CoVirt project on computer security depends on running all applications inside a virtual machine [Chen01]. To keep the system usable in a production environment, we would like the speed of our virtual machine to be within a factor of 2 of a standalone system.

The paper is organized as follows. Section 2 describes two ways to classify virtual machines, focusing on the higher-level interface provided by the VMM and the lower-level platform upon which the VMM is built. Section 3 describes UMLinux, which is the VMM we use in this paper. Section 4 describes a series of extensions to the host operating system that enable virtual machines built on the host operating system to approach the speed of those that run directly on the hardware. Section 5 evaluates the performance benefits

achieved by each host OS extension. Section 6 describes related work, and Section 7 concludes.

2. Virtual machines

Virtual-machine monitors can be classified along many dimensions. This section classifies VMMs along two dimensions: the higher-level interface they provide and the lower-level platform they build upon.

The first way we can classify VMMs is according to how closely the higher-level interface they provide matches the interface of the physical hardware. VMMs such as VM/370 [Goldberg74] for IBM mainframes and VMware ESX Server [Waldspurger02] and VMware Workstation [Sugerman01] for x86 processors provide an abstraction that is identical to the hardware underneath the VMM. Simulators such as Bochs [Boc] and Virtutech Simics [Magnusson95] also provide an abstraction that is identical to physical hardware, although the hardware they simulate may differ from the hardware on which they are running.

Several aspects of virtualization make it difficult or slow for a VMM to provide an interface that is identical to the physical hardware. Some architectures include instructions whose behavior depends on whether the CPU is running in privileged or user mode (sensitive instructions), yet which can execute in user mode without causing a trap to the VMM [Robin00]. Virtualizing these sensitive-but-unprivileged instructions generally requires binary instrumentation, which adds significant complexity and may add significant overhead. In addition, emulating I/O devices at the low-level hardware interface (e.g. memory-mapped I/O) causes execution to switch frequently between the guest operating system accessing the device and the VMM code emulating the device. To avoid the overhead associated with emulating a low-level device interface, most VMMs encourage or require the user to run a modified version of the guest operating system. For example, the VAX VMM security kernel [Karger91], VMware Workstation's guest tools [Sugerman01], and Disco [Bugnion97] all add special drivers in the guest operating system to accelerate the virtualization of some devices. VMMs built on host operating systems often require additional modifications to the guest operating system. For example, the original version of SimOS adds special signal handlers to support virtual interrupts and requires relinking the guest operating system into a different range of addresses [Rosenblum95]; similar changes are needed by User-Mode Linux [Dike00] and UMLinux [Buchacker01].

Other virtualization strategies make the higher-level interface further different from the underlying

hardware. The Denali isolation kernel does not support instructions that are sensitive but unprivileged, adds several virtual instructions and registers, and changes the memory management model [Whitaker02]. Microkernels provide higher-level services above the hardware to support abstractions such as threads and inter-process communication [Golub90]. The Java virtual machine defines a virtual architecture that is completely independent from the underlying hardware.

A second way to classify VMMs is according to the platform upon which they are built [Goldberg73]. *Type I* VMMs such as IBM's VM/370, Disco, and VMware's ESX Server are implemented directly on the physical hardware. *Type II* VMMs are built completely on top of a host operating system. SimOS, User-Mode Linux, and UMLinux are all implemented completely on top of a host operating system. Other VMMs are a hybrid between Type I and II: they operate mostly on the physical hardware but use the host OS to perform I/O. For example, VMware Workstation [Sugerman01] and Connectix VirtualPC [Con01] use the host operating system to access some virtual I/O devices.

A host operating system makes a very convenient platform upon which to build a VMM. Host operating systems provide a set of abstractions that map closely to each part of a virtual machine [Rosenblum95]. A host process provides a sequential stream of execution similar to a CPU; host signals provide similar functionality to interrupts; host files and devices provide similar functionality to virtual I/O devices; host memory mapping and protection provides similar functionality to a virtual MMU. These features make it possible to implement a VMM as a normal user process with very little code.

Other reasons contribute to the attractiveness of using a Type II VMM. Because a Type II VMM runs as a normal process, the developer or administrator of the VMM can use the full power of the host operating system to monitor and debug the virtual machine's execution. For example, the developer or administrator can examine or copy the contents of the virtual machine's I/O devices or memory or attach a debugger to the virtual-machine process. Finally, the simplicity of Type II VMMs and the availability of several good open-source implementations make them an excellent platform for experimenting with virtual-machine services.

A potential disadvantage of Type II VMMs is performance. Current host operating systems do not provide sufficiently powerful interfaces to the bare hardware to support the intensive usage patterns of VMMs. For example, compiling the Linux 2.4.18 kernel inside the UMLinux virtual machine takes 18 times as

long as compiling it directly on a Linux host operating system. VMMs that run directly on the bare hardware achieve much lower performance overhead. For example, VMware Workstation 3.1 compiles the Linux 2.4.18 kernel with only a 30% overhead relative to running directly on the host operating system.

The goal of this paper is to examine and reduce the order-of-magnitude performance overhead associated with running a VMM on a host operating system. We find that a few simple extensions to a host operating system can make it a much faster platform for running a VMM, while preserving the conceptual elegance of the Type II approach.

3. UMLinux

To conduct our study, we use a Type II VMM called UMLinux [Buchacker01]. UMLinux was developed by researchers at the University of Erlangen-Nürnberg for use in fault-injection experiments. UMLinux is a Type II VMM: the guest operating system and all guest applications run as a single process (the *guest-machine process*) on a host Linux operating system. UMLinux provides a higher-level interface to the guest operating system that is similar but not identical to the underlying hardware. As a result, the machine-dependent portion of the guest Linux operating system must be modified to use the interface provided by the VMM. Simple device drivers must be added to interact with the host abstractions used to implement the devices for the virtual machine; a few assembly-language instructions (e.g. `iret` and `in/out`) must be replaced with function calls to emulation code; and the guest kernel must be relinked into a different address range [Hoxer02]. About 17,000 lines of code were added to the guest kernel to work on the new platform. Applications compiled for the host operating system work without modification on the guest operating system.

UMLinux uses functionality from the host operating system that maps naturally to virtual hardware. The guest-machine process serves as a virtual CPU; host files and devices serve as virtual I/O devices; a host TUN/TAP device serves as a virtual network; host signals serve as virtual interrupts; and host memory mapping and protection serve as a virtual MMU. The virtual machine's memory is provided by a host file that is mapped into different parts of the guest-machine process's address space. We store this host file in a memory file system (`ramfs`) to avoid needlessly writing to disk the virtual machine's transient state.

The address space of the guest-machine process differs from a normal host process because it contains

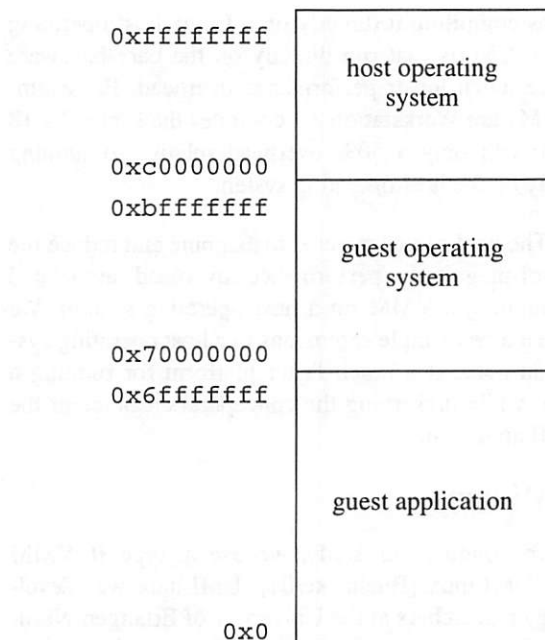


Figure 2: UMLinux address space. As with all Linux processes, the host kernel address space occupies [0xc0000000, 0xffffffff], and the host user address space occupies [0x0, 0xc0000000). The guest kernel occupies the upper portion of the host user space [0x70000000, 0xc0000000), and the current guest application occupies the remainder of the host user space [0x0, 0x70000000).

both the host and guest operating system address ranges (Figure 2). In a standard Linux process, the operating system occupies addresses [0xc0000000, 0xffffffff] while the application is given [0x0, 0xc0000000). Because the UMLinux guest-machine process must hold both the host and guest operating systems, the address space for the guest operating system must be moved to occupy [0x70000000, 0xc0000000), which leaves [0x00000000, 0x70000000) for guest applications. The guest kernel memory is protected using host mmap and munmap system calls. To facilitate this protection, UMLinux maintains a virtual current privilege level, which is analogous to the x86 current privilege level. This is used to differentiate between guest user and guest kernel modes, and the guest kernel memory will be accessible or protected according to the virtual privilege level.

Figure 3 shows the basic system structure of UMLinux. In addition to the guest-machine process, UMLinux uses a VMM process to implement the VMM.

The VMM process serves two purposes: it redirects to the guest operating system signals and system

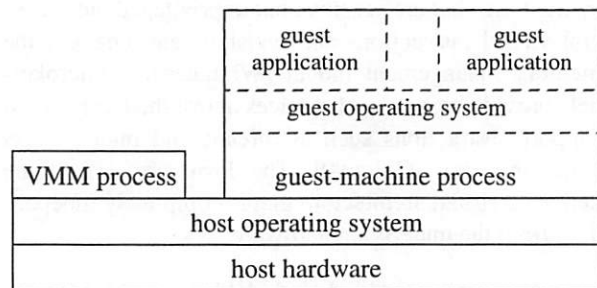


Figure 3: UMLinux system structure. UMLinux uses two host processes. The guest-machine process executes the guest operating system and all guest applications. The VMM process uses ptrace to mediate access between the guest-machine process and the host operating system.

calls that would otherwise go to the host operating system, and it restricts the set of system calls allowed by the guest operating system. The VMM process uses ptrace to mediate access between the guest-machine process and the host operating system. Figure 4 shows the sequence of steps taken by UMLinux when a guest application issues a system call.

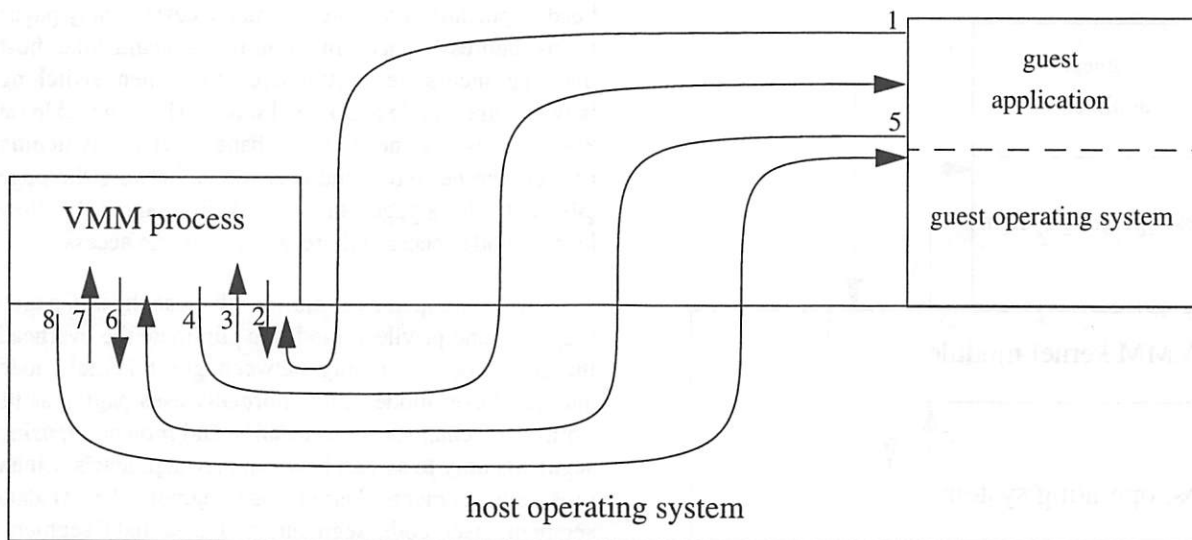
The VMM process is also invoked when the guest kernel returns from its SIGUSR1 handler and when the guest kernel protects its address space from the guest application process. A similar sequence of context switches occurs on each memory, I/O, and timer exception received by the guest-machine process.

4. Host OS support for Type II VMMs

A host operating system makes an elegant and convenient base upon which to build and run a VMM such as UMLinux. Each virtual hardware component maps naturally to an abstraction in the host OS, and the administrator can interact conveniently with the guest-machine process just as it does with other host processes. However, while a host OS provides sufficient functionality to support a VMM, it does not provide the primitives needed to support a VMM *efficiently*.

In this section, we investigate three bottlenecks that occur when running a Type II VMM, and we eliminate these bottlenecks through simple changes to the host OS.

We find that three bottlenecks are responsible for the bulk of the virtualization overhead. First, UMLinux's system structure with two separate host processes causes an inordinate number of context switches on the host. Second, switching between the guest kernel and the guest user space generates a large number of



1. guest application issues system call; intercepted by VMM process via `ptrace`
2. VMM process changes system call to no-op (`getpid`)
3. `getpid` returns; intercepted by VMM process
4. VMM process sends `SIGUSR1` signal to guest `SIGUSR1` handler
5. guest `SIGUSR1` handler calls `mmap` to allow access to guest kernel data; intercepted by VMM process
6. VMM process allows `mmap` to pass through
7. `mmap` returns to VMM process
8. VMM process returns to guest `SIGUSR1` handler, which handles the guest application's system call

Figure 4: Guest application system call. This picture shows the steps UMLinux takes to transfer control to the guest operating system when a guest application process issues a system call. The `mmap` call in the `SIGUSR1` handler must reside in guest user space. For security, the rest of the `SIGUSR1` handler should reside in guest kernel space. The current UMLinux implementation includes an extra section of trampoline code to issue the `mmap`; this trampoline code is started by manipulating the guest machine process's context and finishes by causing a breakpoint to the VMM process; the VMM process then transfers control back to the guest-machine process by sending a `SIGUSR1`.

memory protection operations. Third, switching between two guest application processes generates a large number of memory mapping operations.

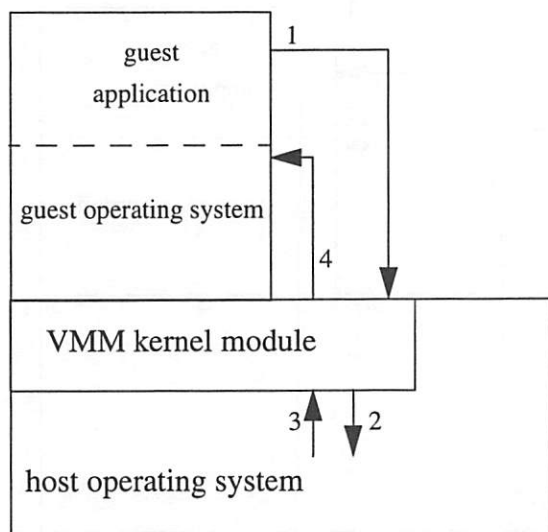
4.1. Extra host context switches

The VMM process in UMLinux uses `ptrace` to intercept key events (system calls and signals) executed by the guest-machine process. `ptrace` is a powerful tool for debugging, but using it to create a virtual machine causes the host OS to context switch frequently between the guest-machine process and the VMM process (Figure 4).

We can eliminate most of these context switches by moving the VMM process's functionality into the host kernel. We encapsulate the bulk of the VMM process functionality in a VMM loadable kernel module. We also modified a few lines in the host kernel's system call and signal handling to transfer control to the VMM

kernel module when the guest-machine process executes a system call or receives a signal. The VMM kernel module and other hooks in the host kernel were implemented in 150 lines of code (not including comments).

Moving the VMM process's functionality into the host kernel drastically reduces the number of context switches in UMLinux. For example, transferring control to the guest kernel on a guest system call can be done in just two context switches (Figure 5). It also simplifies the system conceptually, because the VMM kernel module has more control over the guest-machine process than is provided by `ptrace`. For example, the VMM kernel module can change directly the protections of the guest-machine process's address space, whereas the `ptracing` VMM process must cause the guest-machine process to make multiple system calls to change protections.



1. guest application issues system call; intercepted by VMM kernel module
2. VMM kernel module calls `mmap` to allow access to guest kernel data
3. `mmap` returns to VMM kernel module
4. VMM kernel module sends `SIGUSR1` to guest `SIGUSR1` handler

Figure 5: Guest application system call with VMM kernel module. This picture shows the steps taken by UMLinux with a VMM kernel module to transfer control to the guest operating system when a guest application issues a system call.

4.2. Protecting guest kernel space from guest application processes

The guest-machine process switches frequently between guest user mode and guest kernel mode. The guest kernel is invoked to service guest system calls and other exceptions issued by a guest application process and to service signals initiated by virtual I/O devices. Each time the guest-machine process switches from guest kernel mode to guest user mode, it must first prevent access to the guest kernel's portion of the address space `[0x70000000, 0xc0000000)`. Similarly, each time the guest-machine process switches from guest user mode to guest kernel mode, it must first enable access to the guest kernel's portion of the address space. The guest-machine process performs these address space manipulations by making the host system calls `mmap`, `munmap`, and `mprotect`.

Unfortunately, calling `mmap`, `munmap`, or `mprotect` on large address ranges incurs significant over-

head, especially if the guest kernel accesses many pages in its address space. In contrast, a standalone host machine incurs very little overhead when switching between user mode and kernel mode. The page table on x86 processors need not change when switching between kernel mode and user mode, because the page table entry for a page can be set to simultaneously allow kernel-mode access and prevent user-mode access.

We developed two solutions that use the x86 paged segments and privilege modes to eliminate the overhead incurred when switching between guest kernel mode and guest user mode. Linux normally uses paging as its primary mechanism for translation and protection, using segments only to switch between privilege levels. Linux uses four segments: kernel code segment, kernel data segment, user code segment, and user data segment. Normally, all four segments span the entire address range. Linux normally runs all host user code in CPU privilege ring 3 and runs host kernel code in CPU privilege ring 0. Linux uses the supervisor-only bit in the page table to prevent code running in CPU privilege ring 3 from accessing the host operating system's data (Figure 6).

Our first solution protects the guest kernel space from guest user code by changing the bound on the user code and data segments (Figure 7). When the guest-machine process is running in guest user mode, the VMM kernel module shrinks the user code and data segments to span only `[0x0, 0x70000000)`. When the guest-machine process is running in guest kernel mode, the VMM kernel module grows the user code and data segments to its normal range of `[0x0, 0xffffffff)`. This solution added only 20 lines of code to the VMM kernel module and is the solution we currently use.

One limitation of the first solution is that it assumes the guest kernel space occupies a contiguous region directly below the host kernel space. Our second solution allows the guest kernel space to occupy arbitrary ranges of the address space within `[0x0, 0xc0000000)` by using the page table's supervisor-only bit to distinguish between guest kernel mode and guest user mode (Figure 8). In this solution, the VMM kernel module marks the guest kernel's pages as accessible only by supervisor code (ring 0-2), then runs the guest-machine process in ring 1 while in guest kernel mode. When running in ring 1, the CPU can access pages marked as supervisor in the page table, but it cannot execute privileged instructions (such as changing the segment descriptor). To prevent the guest-machine process from accessing host kernel space, the VMM kernel module shrinks the user code and data segment to span

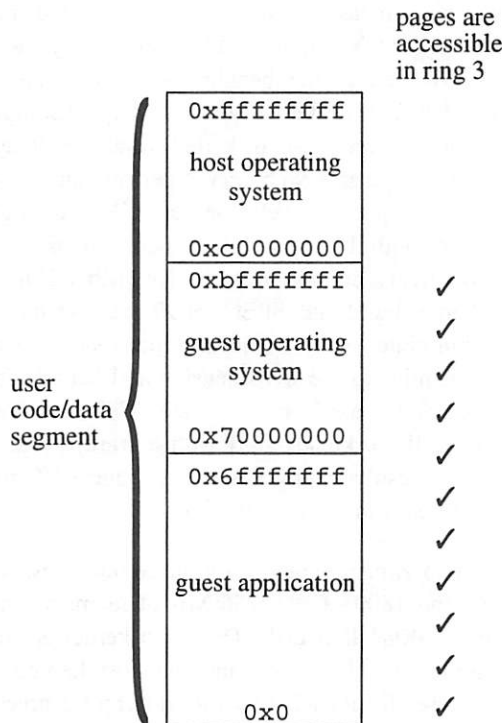


Figure 6: Segment and page protections when running a normal Linux host processes. A normal Linux host process runs in CPU privilege ring 3 and uses the user code and data segment. The segment bounds allow access to all addresses, but the supervisor-only bit in the page table prevents the host process from accessing the host operating system’s data. In order to protect the guest kernel’s data with this setup, the guest-machine process must munmap or mprotect [0x70000000, 0xc0000000) before switching to guest user mode.

only [0x0, 0xc0000000). The guest-machine process runs in ring 3 while in guest user mode, which prevents guest user code from accessing the guest kernel’s data. This allows the VMM kernel module to protect arbitrary pages in [0x0, 0xc0000000) from guest user mode by setting the supervisor-only bit on those pages. It does still require the host kernel and user address ranges to each be contiguous.

4.3. Switching between guest application processes

A third bottleneck in a Type II VMM occurs when switching address spaces between guest application processes. Changing guest address spaces means changing the current mapping between guest virtual pages and the page in the virtual machine’s “physical” memory file. Changing this mapping is done by calling munmap for the outgoing guest application process’s virtual address space, then calling mmap for each resident virtual page

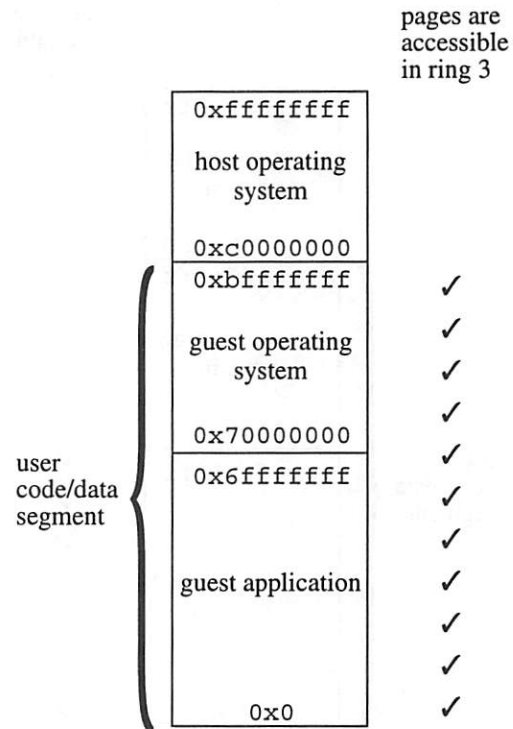


Figure 7: Segment and page protections when running the guest-machine process in guest user mode (solution 1). This solution protects the guest kernel space from guest application processes by changing the bound on the user code and data segments to [0x0, 0x70000000) when running guest user code. When the guest-machine process switches to guest kernel mode, the VMM kernel module grows the user code and data segments to its normal range of [0x0, 0xffffffff].

in the incoming guest application process. UMLinux minimizes the calls to mmap by doing it on demand, i.e. as the incoming guest application process faults in its address space. Even with this optimization, however, UMLinux generates a large number of calls to mmap, especially when the working sets of the guest application processes are large.

To improve the speed of guest context switches, we enhance the host OS to allow a single process to maintain several address space definitions. Each address space is defined by a separate set of page tables, and the guest-machine processes switches between address space definitions via a new host system call switch-guest. To switch address space definitions, switch-guest needs only to change the pointer to the current first-level page table. This task is much faster than mmap’ing each virtual page of the incoming guest application process. We modify the guest kernel to use switch-guest when context switching from one guest application process to another. We reuse initialized

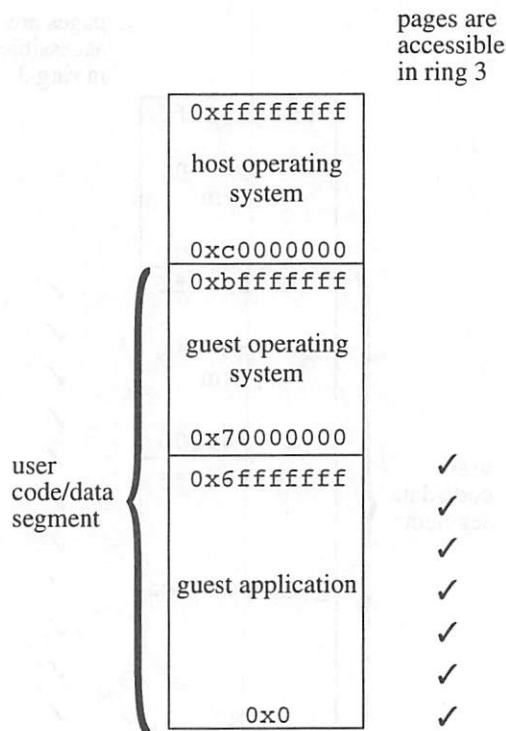


Figure 8: Segment and page protections when running the guest-machine process (solution 2). This solution protects the guest kernel space from guest application processes by marking the guest kernel's pages as accessible only by code running in CPU privilege ring 0-2 and running the guest-machine process in ring 1 when executing guest kernel code. To prevent the guest-machine process from accessing host kernel space, the VMM kernel module shrinks the user code and data segment to span only [0x0, 0xc0000000).

address space definitions to minimize the overhead of creating guest application processes. We take care to prevent the guest-machine process from abusing switchguest by limiting it to 1024 different address spaces and checking all parameters carefully. This optimization added 340 lines of code to the host kernel.

5. Performance results

This section evaluates the performance benefits achieved by each of the optimizations described in Section 4.

We first measure the performance of three important primitives: a null system call, switching between two guest application processes (each with a 64 KB working set), and transferring 10 MB of data using TCP across a 100 Mb/s Ethernet switch. The first two of these microbenchmarks come from the lmbench suite [McVoy96].

We also measure performance on three macrobenchmarks. POV-Ray is a CPU-intensive ray-tracing program. We render the benchmark image from the POV-Ray distribution at quality 8. kernel-build compiles the complete Linux 2.4.18 kernel (make bzImage). SPECweb99 measures web server performance, using the 2.0.36 Apache web server. We configure SPECweb99 with 15 simultaneous connections spread over two clients connected to a 100 Mb/s Ethernet switch. kernel-build and SPECweb99 exercise the virtual machine intensively by making many system calls. They are similar to the I/O-intensive and kernel-intensive workloads used to evaluate Cellular Disco [Govil00]. All workloads start with a warm guest file cache. Each results represents the average of 5 runs. Variance across runs is less than 3%.

All experiments are run on a computer with an AMD Athlon 1800+ CPU, 256 MB of memory, and a Samsung SV4084 IDE disk. The guest kernel is Linux 2.4.18 ported to UMLinux, and the host kernels for UMLinux are all Linux 2.4.18 with different degrees of support for VMMs. All virtual machines are configured with 192 MB of "physical" memory. The virtual hard disk for UMLinux is stored on a raw disk partition on the host to avoid double buffering the virtual disk data in the guest and host file caches and to prevent the virtual machine from benefitting unfairly from the host's file cache. The host and guest file systems have the same versions of all software (based on RedHat 6.2).

We measure baseline performance by running directly on the host operating system (standalone). The host uses the same hardware and software installation as the virtual-machine systems and has access to the full 256 MB of host memory.

We use VMware Workstation 3.1 to illustrate the performance of VMMs that are built directly on the host hardware. We chose VMware Workstation because it executes mostly on host hardware and because it is regarded widely as providing excellent performance. However, note that VMware Workstation may be slower than a Type I VMM that is ideal for the purposes of comparing with UMLinux. First, VMware Workstation issues I/O through the host OS rather than controlling the host I/O devices directly. Second, unlike UMLinux, VMware Workstation can support unmodified guest operating systems, and this capability forces VMware Workstation to do extra work to provide the same interface to the guest OS as the host hardware does. The configuration for VMware Workstation matches that of the other virtual-machine systems, except that VMware

Workstation uses the host disk partition's cacheable block device for its virtual disk.

Figures 9 and 10 summarize results from all performance experiments.

The original UMLinux is hundreds of times slower for null system calls and context switches and is not able to saturate the network. UMLinux is 8x as slow as the standalone host on SPECweb99, 18x as slow as the standalone host on kernel-build and 10% slower than the standalone host on POV-Ray. Because POV-Ray is compute-bound, it does not interact much with the guest kernel and thus incurs little virtualization overhead. The overheads for SPECweb99 and kernel-build are higher because they issue more guest kernel calls, each of which must be trapped by the VMM kernel module and reflected back to the guest kernel by sending a signal.

VMMs that are built directly on the hardware execute much faster than a Type II VMM without host OS support. VMware Workstation 3.1 executes a null system call nearly as fast as the standalone host, can saturate the network, and is within a factor of 5 of the context switch time for a standalone host. VMware Workstation 3.1 incurs an overhead of 6-30% on the intensive macrobenchmarks (SPECweb99 and kernel-build).

Our first optimization (Section 4.1) moves the VMM functionality into the kernel. This improves performance by a factor of about 2-3 on the microbenchmarks, and by a factor of about 2 on the intensive macrobenchmarks.

Our second optimization (Section 4.2) uses segment bounds to eliminate the need to call `mmap`, `munmap`, and `mprotect` when switching between guest kernel mode and guest user mode. Adding this optimization improves performance on null system calls and context switches by another factor of 5 (beyond the performance with just the first optimization) and enables UMLinux to saturate the network. Performance on the two intensive macrobenchmarks improves by a factor of 3-4.

Our final optimization (Section 4.3) maintains multiple address space definitions to speed up context switches between guest application processes. This optimization has little effect on benchmarks with only one main application process, but it has a dramatic affect on benchmarks with more than one main application process. Adding this optimization improves the context switch microbenchmark by a factor of 13 and improves kernel-build by a factor of 2.

With all three host OS optimizations to support VMMs, UMLinux runs all macrobenchmarks well within our performance target of a factor of 2 relative to standalone. POV-Ray incurs 1% overhead; kernel-build incurs 35% overhead; and SPECweb99 incurs 14% overhead. These overheads are comparable to those attained by VMware Workstation 3.1.

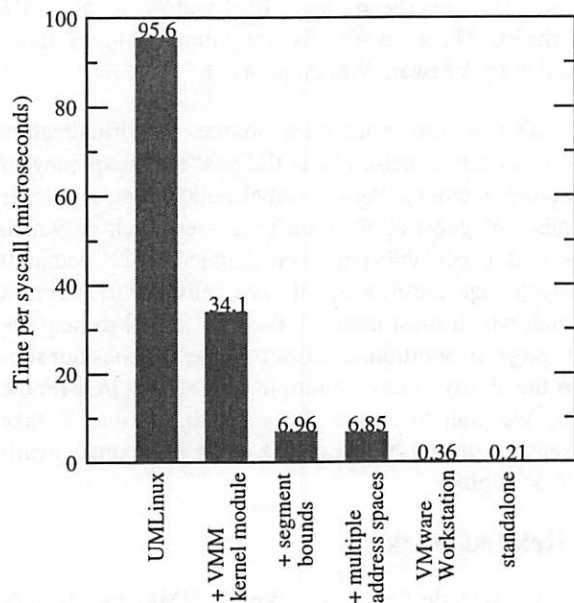
The largest remaining source of virtualization overhead for kernel-build is the cost and frequency of handling memory faults. kernel-build creates a large number of guest application processes, each of which maps its executable pages on demand. Each demand-mapped page causes a signal to be delivered to the guest kernel, which must then ask the host kernel to map the new page. In addition, UMLinux currently does not support the ability to issue multiple outstanding I/Os on the host. We plan to update the guest disk driver to take advantage of non-blocking I/O when it becomes available on Linux.

6. Related work

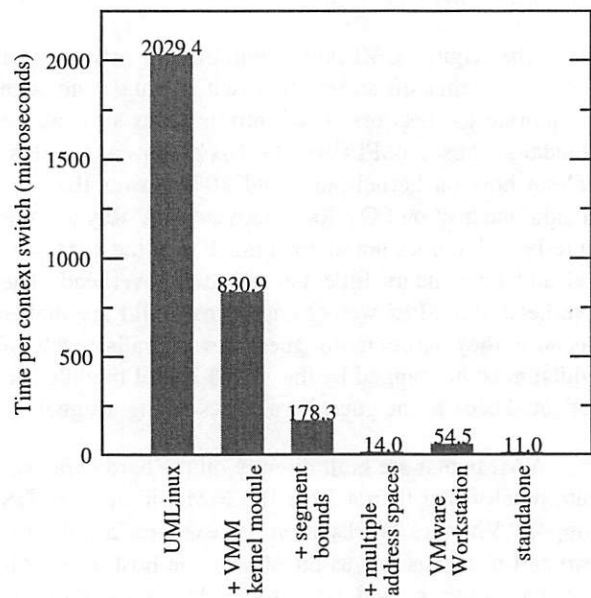
User-Mode Linux is a Type II VMM that is very similar to UMLinux [Dike00]. Our discussion of User-Mode Linux assumes a configuration that protects guest kernel memory from guest application processes (jail mode). The major technical difference between the User-Mode Linux and UMLinux is that User-Mode Linux uses a separate host process for each guest application process, while UMLinux runs all guest code in a single host process. Assigning each guest application process to a separate host process technique speeds up context switches between guest application processes, but it leads to complications such as keeping the shared portion of the guest address spaces consistent and difficult synchronization issues when switching guest application processes [Dike02a].

User-Mode Linux in jail mode is faster than UMLinux (without host OS support) on context switches (157 vs. 2029 microseconds) but slower on system calls (296 vs. 96 microseconds) and network transfers (54 vs. 39 seconds). User-Mode Linux in jail mode is faster on kernel-build (1309 vs. 2294 seconds) and slower on SPECweb99 (200 vs. 172 seconds) than UMLinux without host OS support.

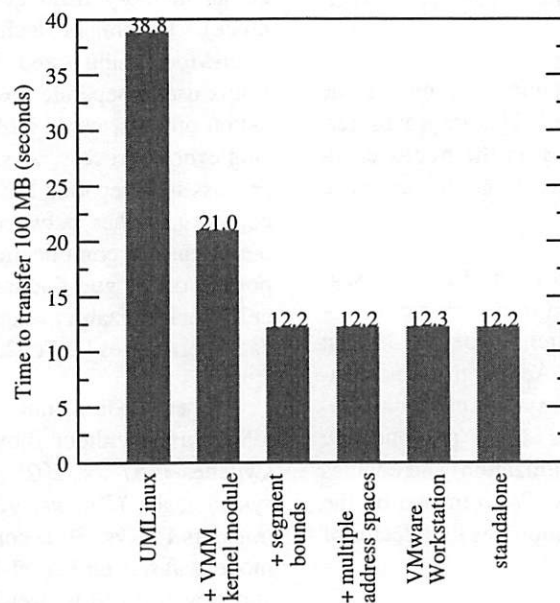
Concurrently with our work on host OS support for VMMs, the author of User-Mode Linux proposed modifying the host OS to support multiple address space definitions for a single host process [Dike02a]. Like the optimization in Section 4.3, this would speed up switches between guest application processes and allow User-Mode Linux to run all guest code in a single host



null system call

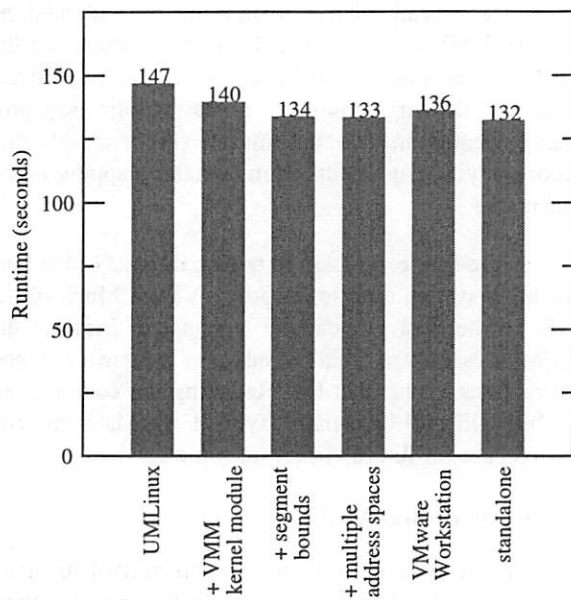


context switch

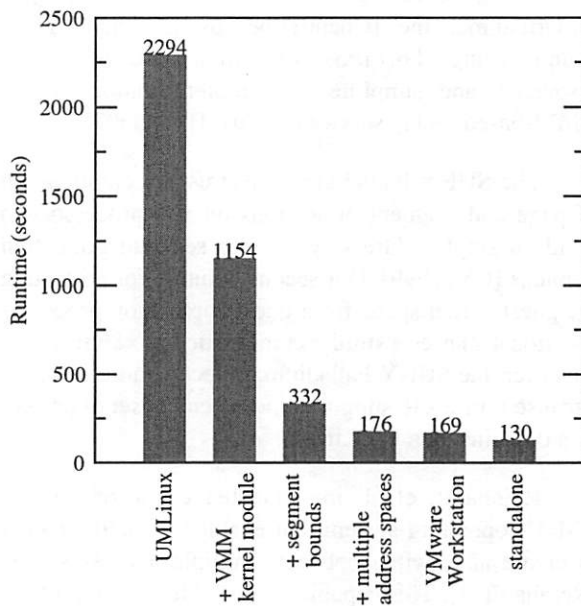


network transfer

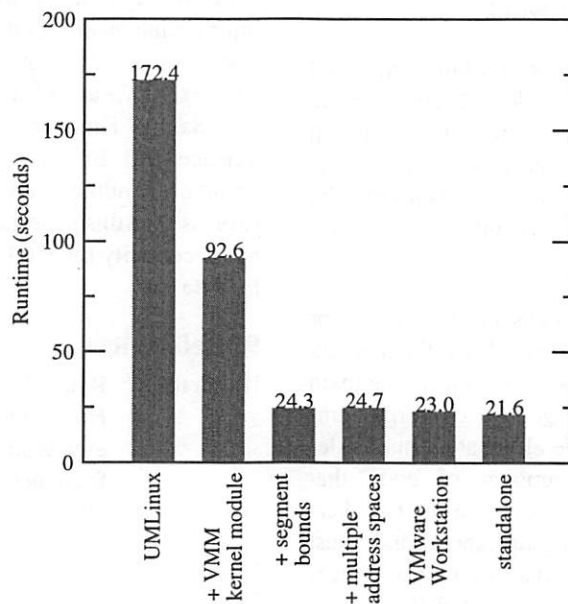
Figure 9: Microbenchmark results. This figure compares the performance of different virtual-machine monitors on three microbenchmarks: a null guest system call, context switching between two 64 KB guest application processes, and receiving 10 MB of data over the network. The first four bars represent the performance of UMLinux with increasing support from the host OS. Each optimization level is **cumulative**, i.e. it includes all optimizations of the bars to the left. The performance of a standalone host (no VMM) is shown for reference. Without support from the host OS, UMLinux is much slower than a standalone host. Adding three extensions to the host OS improves the performance of UMLinux dramatically.



POV-Ray



kernel-build



SPECweb99

Figure 10: Macrobenchmark results. This figure compares the performance of different virtual-machine monitors on three macrobenchmarks: the POV-Ray ray tracer, compiling a kernel, and SPECweb99. The first four bars represent the performance of UMLinux with increasing support from the host OS. Each optimization level is **cumulative**, i.e. it includes all optimizations of the bars to the left. The performance of a standalone host (no VMM) is shown for reference. Without support from the host OS, UMLinux is much slower than a standalone host. Adding three extensions to the host OS allows UMLinux to approach the speed of a Type I VMM.

process. Implementation of this optimization is currently underway [Dike02b], though User-Mode Linux still uses two separate host processes, one for the guest kernel and one for all guest application processes. We currently use UMLinux for our CoVirt research project on virtual machines [Chen01] because running all guest code in a single host process is simpler, uses fewer host resources, and simplifies the implementation of our VMM-based replay service (ReVirt) [Dunlap02].

The SUNY Palladium project used a combination of page and segment protections on x86 processors to divide a single address space into separate protection domains [Chiueh99]. Our second solution for protecting the guest kernel space from guest application processes (Section 4.2) uses a similar combination of x86 features. However, the SUNY Palladium project is more complex because it needs to support a more general set of protection domains than UMLinux.

Reinhardt, et al. implemented extensions to the CM-5's operating system that enabled a single process to create and switch between multiple address spaces [Reinhardt93]. This capability was added to support the Wisconsin Wind Tunnel's parallel simulation of parallel computers.

7. Conclusions and future work

Virtual-machine monitors that are built on a host operating system are simple and elegant, but they are currently an order of magnitude slower than running outside a virtual machine, and much slower than VMMs that are built directly on the hardware. We examined the sources of overhead for a VMM that run on a host operating system.

We found that three bottlenecks are responsible for the bulk of the performance overhead. First, the host OS required a separate host user process to control the main guest-machine process, and this generated a large number of host context switches. We eliminated this bottleneck by moving the small amount of code that controlled the guest-machine process into the host kernel. Second, switching between guest kernel and guest user space generated a large number of memory protection operations on the host. We eliminated this bottleneck in two ways. One solution modified the host user segment bounds; the other solution modified the segment bounds and ran the guest-machine process in CPU privilege ring 1. Third, switching between two guest application processes generated a large number of memory mapping operations on the host. We eliminated this bottleneck by allowing a single host process to maintain several address space definitions. In total, 510 lines of

code were added to the host kernel to support these three optimizations.

With all three optimizations, performance of a Type II VMM on macrobenchmarks improved to within 14-35% overhead relative to running on a standalone host (no VMM), even on benchmarks that exercised the VMM intensively. The main remaining source of overhead was the large number of guest application processes created in one benchmark (kernel-build) and accompanying page faults from demand mapping in the executable.

In the future, we plan to reduce the size of the host operating system used to support a VMM. Much of the code in the host OS can be eliminated, because the VMM uses only a small number of system calls and abstractions in the host OS. Reducing the code size of the host OS will help make Type II VMMs a fast and trusted base for future virtual-machine services.

8. Acknowledgments

We are grateful to the researchers at the University of Erlangen-Nürnberg for writing UMLinux and sharing it with us. In particular, Kerstin Buchacker and Volkmar Sieh helped us understand and use UMLinux. Our shepherd Ed Bugnion and the anonymous reviewers helped improve the quality of this paper. This research was supported in part by National Science Foundation grants CCR-0098229 and CCR-0219085 and by Intel Corporation. Samuel King was supported by a National Defense Science and Engineering Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

9. References

- [Barnett02] Ryan C. Barnett. Monitoring VMware Honeypots, September 2002. http://honeypots.sourceforge.net/monitoring_vmware_honeypots.html.
- [Boc] <http://bochs.sourceforge.net/>.
- [Bressoud96] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.
- [Buchacker01] Kerstin Buchacker and Volkmar Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings of the 2001 IEEE Symposium on High Assurance System*

- Engineering (HASE)*, pages 95–105, October 2001.
- [Bugnion97] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.
- [Chen01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, May 2001.
- [Chiueh99] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions. In *Proceedings of the 1999 Symposium on Operating Systems Principles*, December 1999.
- [Con01] The Technology of Virtual Machines. Technical report, Connectix Corp., September 2001.
- [Dike00] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, October 2000.
- [Dike02a] Jeff Dike. Making Linux Safe for Virtual Machines. In *Proceedings of the 2002 Ottawa Linux Symposium (OLS)*, June 2002.
- [Dike02b] Jeff Dike. User-Mode Linux Diary, November 2002. <http://user-mode-linux.sourceforge.net/diary.html>.
- [Dunlap02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002.
- [Goldberg73] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, February 1973.
- [Goldberg74] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, June 1974.
- [Golub90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the 1990 USENIX Summer Conference*, 1990.
- [Govil00] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):226–262, August 2000.
- [Hoxer02] H. J. Hoxer, K. Buchacker, and V. Sieh. Implementing a User-Mode Linux with Minimal Changes from Original Kernel. In *Proceedings of the 2002 International Linux System Technology Conference*, pages 72–82, September 2002.
- [Karger91] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11), November 1991.
- [Magnusson95] Peter Magnusson and B. Werner. Efficient Memory Simulation in SimICS. In *Proceedings of the 1995 Annual Simulation Symposium*, pages 62–73, April 1995.
- [McVoy96] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the Winter 1996 USENIX Conference*, January 1996.
- [Meushaw00] Robert Meushaw and Donald Simard. NetTop: Commercial Technology in High Assurance Applications. *Tech Trend Notes: Preview of Tomorrow's Information Technologies*, 9(4), September 2000.
- [Nieh00] Jason Nieh and Ozgur Can Leonard. Examining VMware. *Dr. Dobbs's Journal*, August 2000.
- [Reinhardt93] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel Support for the Wisconsin Wind Tunnel. In *Proceedings of the 1993 Usenix Symposium on Microkernels and Other Kernel Architectures*, pages 73–89, September 1993.
- [Robin00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 2000 USENIX Security Symposium*, August 2000.
- [Rosenblum95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: the SimOS approach. *IEEE Parallel &*

Distributed Technology: Systems & Applications, 3(4):34–43, January 1995.

[Sugerman01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Technical Conference*, June 2001.

[Waldspurger02] Carl A. Waldspurger. Memory Resource Management in VMware ESX

Server. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[Whitaker02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

A Multi-User Virtual Machine

Grzegorz Czajkowski

Laurent Daynès

Ben Titzer

*Sun Microsystems Laboratories
2600 Casey Avenue
Mountain View, CA 94043, USA*

*S³ Lab, Purdue University
1398 Computer Sciences Bldg.
West Lafayette, IN 47906, USA*

grzegorz.czajkowski@sun.com

laurent.daynes@sun.com

titzer@purdue.edu

ABSTRACT

Recent efforts aimed at improving the scalability of the Java™ platform have focused primarily on the safe collocation of multiple applications in the virtual machine. This is often beneficial for various performance metrics, but ultimately leads to a *single-user* multitasking environment. The lack of multi-user capabilities forms a barrier to the scalability of multitasking virtual machines, as it requires one per user. In this paper we demonstrate how to enhance a multitasking virtual machine with multi-user support. In particular, users can securely manipulate their private files, load their own native libraries without endangering other computations, and use all standard APIs. Auxiliary processes are needed to provide multiple operating system resource and user contexts, but no modifications are needed to the operating system itself.

1 INTRODUCTION

Program execution environments based on safe languages have become an important part of the computing landscape, as demonstrated by a growing number of middleware systems taking advantage of the Java platform [GJS+00]. The scalability of the underlying virtual machines is key to efficient resource utilization and consequently to widespread acceptance of safe languages. Several recent projects have demonstrated that scalability can be improved by re-architecting the run-time system or by program transformations that enable execution of multiple applications in a single instance of the virtual machine with certain degrees of application isolation [HCC+98, BV99, BHL00, CD01]. Although the results of these efforts differ considerably with respect to features available and performance, they are invariably multitasking *single-user* environments.

The single-user behavior manifests itself in several ways. First, since the run-time system executes as a single operating system (OS) process, with a single set of user privileges and session attributes, private files of only the user whose effective user id the virtual machine process has can be accessed. Typically 'running as root' to address this problem is not an

option, as it is dangerous from the security viewpoint. Setting the effective user id affects the whole process, which would only be correct if all file access operations performed by the virtual machine were serialized. The second issue concerns user-supplied native libraries. Most multitasking systems based on safe languages do not allow such code, since a malfunction might jeopardize the whole environment and all the applications in it. Finally, the correct execution of certain core components of the safe language platform, such as the windowing subsystem, is not guaranteed in presence of multiple users, or even when a single user runs multiple applications that require such components. The issue here is the interference of unrelated computations via global state of core native libraries.

Our previous work has demonstrated that collocating of computations in a multitasking virtual machine combined with aggressive sharing of run-time data structures can improve performance and significantly decrease start-up time and memory footprint [CD01]. To fully realize its potential, this approach must address the multi-user issues mentioned above. A case in point is thin-client environments where stateless desktop consoles access a shared pool of computational resources in one or more powerful servers [SLN99]. At peak times the number of active users on a single Sun Ray™ installation can reach hundreds. If every user runs just a single application in a dedicated Java virtual machine (JVM™), the combined resource requirements severely stress the system and negatively impact the user experience. The bottom line is that a single multi-user multitasking virtual machine offers the potential to utilize resources better than a collection of single-user multitasking virtual machines, just as a single-user multitasking virtual machine scales better than a collection of virtual machines each executing a single application.

This paper demonstrates that it is possible to construct a complete and fully compliant multitasking multi-user virtual machine for executing programs written in the Java™ programming language. Our solutions take advantage of the existing OS infrastructure and do not require high engineering effort. In particular, we

enhance the Multitasking Virtual Machine (or MVM) [CD01] with the ability to execute applications of different users. This includes correct maintaining of user identity, including access control of users' private files, the ability to run user-supplied native code safely, and a mechanism to ensure correct operation of core native libraries in the presence of multiple users. The last feature is based on a novel technique that transparently replicates the global state of shared libraries. The first two enhancements take advantage of the ability to pass open file descriptors among processes and improve on MVM's ability to isolate native code.

The rest of the paper is structured as follows. Section 2 contains an overview of the architecture, Sections 3-5 describe the handling of user identity, user-supplied native code, and core native libraries, respectively, along with performance details. A discussion of design alternatives and related work is given in Section 6.

2 ARCHITECTURE OVERVIEW

The proposed multi-user virtual machine architecture, dubbed MVM-2, is described later on, after an introduction to MVM (this term will consistently refer to the previous version of the system [CD01]).

2.1 Background on MVM

MVM is a general-purpose virtual machine for executing multiple applications, written in the Java programming language, on behalf of a single user. It is based on the Java HotSpot™ virtual machine (referred to from now on as HSVM) [Sun00a] and its client compiler, version 1.3.1 for the Solaris™ Operating Environment [MM01]. In experiments described in the following three sections version 2.9 of the operating environment was used, running on a Sun Enterprise™ 3500 server with four UltraSPARC™ II processors and 4GB of main memory.

Applications executing in MVM are referred to as *isolates* [JCP01]. MVM-aware applications, such as application server engines, can use the provided API to control the life-cycle (e.g., creation and asynchronous termination) of other isolates. The main (first) isolate does not have to be a server – it can be any application written in the Java programming language. A simple example of the API is the creation of an isolate, which will execute `MyClass` with a single argument “abc”:

```
new Isolate("MyClass", new String[] {"abc"}).start(...);
```

The key design principle of MVM was to examine each component of the JVM and determine whether sharing it among isolates can lead to any interference among them. Non-shareable components are either replicated on a per-isolate basis or made *isolate re-entrant*, that is, usable by many isolates without

causing any inter-isolate interference. They include static fields, class initialization state, and instances of `java.lang.Class`.

Shareable components that require modification to become isolate re-entrant include the constant pool, the interpreter, the dynamic compiler, and the code it produces. An arbitrary number of isolates in MVM can share the code (bytecode and compiled) of both core and application classes. Runtime modifications make the replication of non-shareable components transparent. In effect, each application “believes” it executes in its own private JVM, as there is no interference due to mutable run-time data structures visible directly or indirectly by the application code. Similarly, certain Java Development Kit (JDK™) classes, such as `System` and `Runtime` had to be modified to make operations such as `System.exit()` apply only to the calling isolate.

The heaps of isolates are logically disjoint. The separation of isolates' data sets in MVM implies that isolates cannot directly share objects, and the only way for isolates to communicate is to use copying communication mechanisms, either standard ones, such as sockets, or low-level custom protocols [PCD+02]. Another option is to use *links*, which are a low-level isolate-to-isolate communication mechanism introduced by the isolate API [JCP01].

In MVM most of the class representation is shared, and so is the class loading, linking, and run-time compilation effort. In particular, only when a class is loaded into MVM for the first time the actual file fetching, parsing, verification, building of a main-memory run-time representation of the class, and several other steps are performed. They do not need to be repeated when another isolate uses the same class.

2.2 Process Model

Essential to bringing multi-user capabilities to MVM is a process model that encapsulates the ideas of protection and access control. In our design, a single instance of MVM-2 exists as one process. It contains multiple isolates. Isolates may be started within the JVM by different users through a separate login program called *Jlogin*, written in C. *Jlogin* corresponds to a notion of a user session, and is used to start a single isolate – the user simply types in the name of the main class and its arguments, similarly to running the standard “java” command.

After session initialization, *Jlogin* serves as a daemon process that services certain requests issued by the initial isolate in the session; these requests are related to (i) user identity – this breaks down into several sub-cases: accessing the file system, accessing environment information related to the user session (i.e., an instance of *Jlogin*), and maintaining user

identity and session attributes across process creation via the `Runtime.exec()` method, and (ii) interacting with user-supplied native code. The Jlogin process has the effective user id and associate privileges of the actual user, regardless of process attributes of MVM-2 (Figure 1).

The initial isolate has its standard input/output/error streams connected appropriately to its Jlogin process; these streams may be shared with descendent isolates (isolates created in the course of the program execution, and not by spawning another Jlogin process) if the initial one sets them so. Each isolate, regardless of how it was created, has its own instance of Jlogin (lazily created for non-initial isolates), so that a failure of native code associated with one isolate does not affect the others. Optimizations to this basic scheme, such as optional association of multiple isolates with a single Jlogin process, are not further pursued in this paper.

Each user can have multiple sessions. To simplify further discussion let us assume that the initial process does not spawn further isolates and that there is only one Jlogin per user.

The first isolate of MVM-2 is a simple application called Mserver that listens on a socket for connections from Jlogin processes. Each new Jlogin connects to Mserver and the two exchange information such as relevant environment variables and user settings. Jlogin then sends a request to Mserver to create an isolate to run the application the user specified when starting that Jlogin instance. The isolate connects to its Jlogin's standard input, output, and error streams.

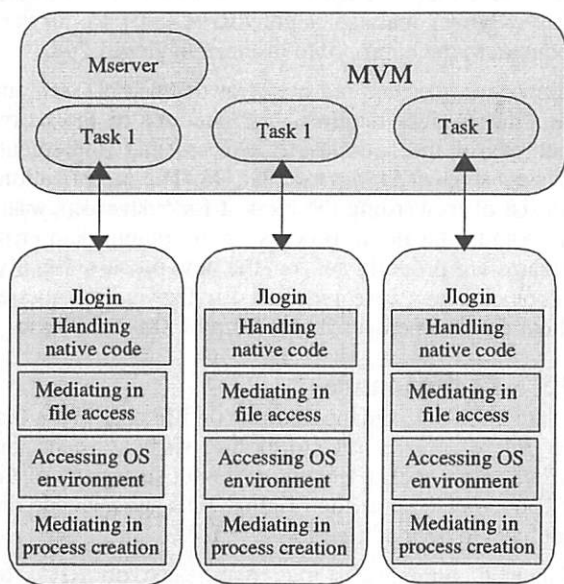


Figure 1. Three users execute applications in MVM-2; each of them has an instance of the Jlogin process.

Multiple Jlogin processes from different users can connect to the Mserver to launch their applications within the same instance of MVM-2.

3 USER IDENTITY

In MVM-2 the user identity and session attributes of Jlogin are associated with isolates. In particular, users are able to access their private files securely, and at the same time are not able to elevate their own privilege or circumvent the OS access control mechanisms. Moreover, user identity is properly preserved across process creation. These issues are discussed below.

3.1 File Access

A straightforward approach to enable secure file access for multiple users might be to modify the JVM so that it performs explicit access control checks for file operations that are at least as restrictive as the underlying OS. Upon a user request, the JVM would first decide whether a file system operation was permitted for that user and if so, perform that operation. This scheme would work successfully if the JVM itself had the right to perform all the operations requested by all users – if it was running as root on a UNIX^(R) system, for example. However, running the JVM itself with the highest privilege could damage the system if the JVM itself crashes or is compromised. Moreover, properly emulating the underlying system's access rights checks seems quite complex.

To address these issues we have designed the Remote File Access (RFA) subsystem of MVM-2. RFA forwards certain file system operations, such as file opening or deleting, to the Jlogin process associated with the requesting isolate. To accomplish this, open file descriptors must be passed between the MVM-2 and Jlogin processes. This feature is available in several UNIX inter-process communication (IPC) mechanisms, such as unnamed stream pipes, UNIX domain sockets, or streams [Stev90]. Our implementation of RFA uses doors [MM01], a high-performance IPC mechanism available on the Solaris Operating Environment. Operations such as read, write, seek, and close need not be done remotely, since they are just operations on opened file descriptors local to the virtual machine and are not subject to access control checks. Thus such performance-critical operations as reading, writing, and seeking do not incur the cost of IPC. Doors allow for examining clients' credentials. In particular, the process id of the caller can be obtained. This information is used by Jlogin to verify that RFA requests are issued by the given instance of MVM-2.

As Jlogin runs with the access privileges of the user who started the isolate, the OS access control mechanism for this process enforces the correct privilege level for the corresponding isolate. In this

way, no elevation of privilege occurs and access to the user's private files is permitted correctly, in accordance with the UNIX access control semantics. It is not required that MVM-2 run at the highest privilege level. It can be started by any non-privileged user. The Java security model is unmodified – appropriate permissions (i.e., instances of `java.io.FilePermission`) are still a necessary prerequisite to file access.

As MVM-2 is JDK 1.3.1-compliant, it does not have the New I/O (NIO) APIs [Sun02], introduced in JDK 1.4. One of the classes defined there, `java.nio.channels.FileChannel`, allows application code to create file locks held on behalf of the entire JVM (e.g., implemented through the `fcntl()` system call). This is a potential interference point for multiple computations collocated in the same instance of the JVM. Similarly to how MVM-2 gives each isolate secure access to all of the corresponding user's files, a 1.4-compliant MVM-2 would forward file locking operations to Jlogin to provide locking semantics indistinguishable from a model in which each application executes in a dedicated virtual machine process.

The commands implemented in the RFA protocol are the following: `RFA_open` (open a file given a pathname), `RFA_mkdir` (create a directory), `RFA_mode` (get the access mode bits), `RFA_getmtime` (get the modified time of a pathname), `RFA_length` (get the length of a file), `RFA_access` (check access to a given pathname), `RFA_list` (return a list of files in a given directory), `RFA_setmtime` (change the modified time of a file), `RFA_remove` (remove a given pathname), `RFA_protect` (write-protect a given pathname), and `RFA_rename` (rename a given pathname). These commands form a basic set of operations needed to implement the semantics of the `java.io` classes that operate on files.

3.2 Process Creation

Another issue related to user identity is creating new processes. The Java programming language allows applications to execute arbitrary programs as new processes via the `exec()` method of the `java.lang.Runtime` class, and to kill or wait for these processes via the methods of the `java.lang.Process` class. Standard implementations of the JVM utilize the `fork/exec` capabilities of the underlying OS to provide this functionality.

MVM-2 ensures that a process started by an application running on behalf of a particular user inherits not just the privilege of that user, but also the environment of the process that launched that application. Running MVM-2 with root privileges and then “fork-exec-ing” it combined with using `setuid()` would not adequately address this issue, as the environment attributes would be of the MVM-2

process and not of the appropriate Jlogin process. Our solution is similar to the way MVM-2 deals with file accesses: requests to spawn a new process, to wait for its completion, or to kill it are forwarded to Jlogin of the current isolate. This guarantees that the new processes runs with both the correct user identity and inherits the appropriate environment properties (e.g., current directory, environment variables, etc.).

3.3 Accessing Environment Properties

Even though the Java platform does not provide an API to perform an equivalent of `getenv()` and `setenv()` available on the UNIX platforms, internally the JDK accesses the environment, for example to obtain the values of the `TZ` (time zone) and `DISPLAY` variables. MVM-2 forwards these requests to Jlogin.

3.4 Changes to the JVM and the JDK

RFA required only one minor change to HSVM: the file opening operation had to be modified to select the correct Jlogin process to forward the RFA request to. This operation, used internally by the native code of some core classes, is encapsulated in the internal JVM file opening call, which ultimately calls the OS.

Changes were required to classes in the `java.io` package. The non-public abstract class `FileSystem` encapsulates the functionality of the file system for other classes in the package. It has a static method used to retrieve an object that represents the correct underlying file system, for example an instance of `UnixFileSystem`. The only change to `FileSystem` was to make this static method return an instance of `RFAFileSystem` instead. `RFAFileSystem` extends `UnixFileSystem` making remote RFA calls to forward requests to the appropriate instance of Jlogin.

Supporting the required behavior of `Runtime.exec()` and the `Process` class requires modifications of the native methods of the sub-class of `Process` that implements process support for a particular OS. The modifications consist of forwarding the request for `fork()/exec()`, `wait()`, and `kill()` to the Jlogin process. Input, output, and error streams are properly set for the new process. Finally, minor changes were required for forwarding queries about environment attributes from MVM-2 to Jlogin.

3.5 Performance

There is no performance impact on file operations that do not require an IPC to Jlogin, such as reads and writes. Also, socket operations do not suffer any of the IPC overhead, since the original JDK `java.net` code did not have to be modified for MVM-2.

Table 1 summarizes the overheads of RFA on `java.io` file operations that need to be mediated by Jlogin. The additional cost is between 81% for `mkdir()` and 268% for `lastModified()`. The variance in relative

Operation	Overhead
open()	261%
length()	257%
lastModified()	268%
renameTo()	125%
setReadOnly()	146%
isFile()	178%
canWrite()	242%
list()	141%
mkdir()	81%

Table 1. RFA overheads on java.io.File operations.

and MVM-2, they still need to be forwarded to Jlogin. Otherwise, insufficient level of privilege may prevent, for example, listing of a directory.

The impact of these operations on the actual file manipulation depends on how many of them are issued relative to the number of reads and writes. For instance, opening a file (remote operation) followed by a hundred 80-byte `FileOutputStream.write(byte[])` operations and a close (all local operations) is 27% more expensive than when run with an unmodified HSVM, while the same sequence with a thousand writes is 2.8% more expensive. Another micro-example can be the creation of a properties object from a file:

```
new Properties().load(new FileInputStream(fileName));
```

The overheads depend on the size of the named file. For the *flavormap.properties* distributed with the JDK 1.3.1 (size: 929 bytes) the overhead when compared to HSVM is 18%. For *psfontj2d.properties* (size: 10669 bytes) the overhead is 7.7%.

How this translates into application execution time overheads depends on the intensity of using RFA-mediated file operations. For example, the performance impact of RFA on the file system intensive *javac* benchmark from the SpecJVM98 benchmark suite [Spec98], is less than 0.5%.

4 USER-SUPPLIED NATIVE CODE

The coexistence of programs written in a safe language with user-supplied, unsafe (native) code is convenient, as it enables direct access to hardware, OS resources, and legacy code and can improve performance. But the inherent lack of memory safety in native code may break the contract offered by a safe language. In the case of a single application executing in the JVM, a bug in an application (user-level) native library will disrupt or abnormally terminate this

particular application only. The consequence of an errant native library carelessly loaded into MVM-2 can be much more serious. In addition to causing the loading application to malfunction, such a library may corrupt the data of other applications, perform arbitrary operations with the privilege level of the virtual machine, or crash the whole virtual machine, causing denial of service.

Memory safety is not the only issue, though. Guaranteeing the conflict-free use of system resources by the JVM and native code is equally important. Native code is written against two interfaces: the Java Native Interface (JNI) [Lian99], which is the sole interaction point between the JVM and the application, and the host OS interfaces, involving the usual libraries for I/O, threading, math, networking, etc. The latter is also the interface against which the JVM is written. The problem is that the JVM makes certain decisions regarding the use of the host OS interface and of available resources, and these decisions may conflict with their use by the user-supplied native code. For example, signal handlers may have to be instantiated to handle exceptions that are part of the operation of the JVM (e.g., to detect memory access and arithmetic exceptions). Another example is the JVM's choice of a memory management regime for purposes such as the allocation of thread stacks. In each of these cases an arbitrary use of any of these resources by user-supplied native code can cause the virtual machine to malfunction.

MVM dealt with these issues by automatically and transparently executing user-supplied native libraries in a separate process. Each isolate that needs it and has the necessary permissions has one such process. This means that the only interface between the JVM and native libraries becomes JNI. There is then no implicit contract concerning memory management, threading, signal handling, and other issues. This refactoring solves the composability problem neatly. The native code in a separate process has full control of its own resources. There are no unexpected interactions with MVM via memory, signals, threads, and so on. However, the design of MVM's native code isolation was not suitable for multiple users. The problems are described in Sec. 4.2, which is followed by the description of the new design. First, the essential information on JNI is given.

4.1 JNI Essentials

JNI interacts with the JVM via *downcalls* (when a Java method calls a native method) and *upcalls* (when a native method requests a service from the JVM). Upcalls enable accessing static and instance fields and array elements, invoking methods, entering and exiting monitors, creating new objects, using reflection, and throwing and catching exceptions.

Downcalls result in calls to C or C++ functions, whose names are generated by the *javah* tool from the names of methods declared as native. The naming convention is `Java_packageName_className_methodName`. An optional signature may also be appended to the end of the name to support C++ or to disambiguate overloaded method names. The JVM uses this naming convention to bind the address of an exported function to that of the native method at invocation time.

Upcalls are invoked via a *JNI environment* interface, a pointer to which is always passed as the first argument to all JNI upcalls and downcalls. Objects, classes, fields, and methods are never accessed directly, but rather via appropriate opaque references or identifiers. These references are meaningful only to the JNI functions, and shield native code from the details of particular implementations of JNI.

4.2 Previous Native Code Isolation

Native code isolation (NCI) is achieved in MVM by generating ahead of time a proxy library for each specified library holding native methods. For each function in the original library there is a function with the same name in the corresponding proxy library. The JVM's environment variable `LD_LIBRARY_PATH` is manipulated such that the virtual machine will load proxy libraries with the same name; the original (unmodified) libraries are loaded into a remote NCI process. From that point on, the proxy library and the NCI process orchestrate, transparently to the original native code, the forwarding of native method invocations and JNI upcalls across process boundaries.

The functions in proxy libraries are redefined to forward their arguments, along with information uniquely identifying the function, to a dedicated, transparently created process. Upon receipt of such a request, the process executes the required function with the supplied arguments. Just before the execution of the function, the first received argument is replaced with a custom JNI environment pointer. This custom JNI environment redefines all JNI upcalls so that each of them ships all of its arguments along with its unique identifier back to MVM, where the upcall is dispatched to the JVM's actual implementation of the JNI call. Upcalls are always executed in the same thread that issued the original downcall. For instance, an exception thrown in an upcall has to be dispatched to the thread that caused the downcall.

However, the proxy library approach proved to be inflexible for MVM-2 because: (i) it requires manual intervention to generate the proxy libraries before executing the application, (ii) loading multiple libraries in the same remote process is not dynamic, (iii) allocation of method identifiers is static and not unique across the JVM, (iv) the proxy library would

have to be redesigned to dispatch to different instances of Jlogin based on an isolate identifier.

4.3 New Design

As in previous design, in MVM-2 user-supplied native libraries are loaded into a separate process – in our current implementation Jlogin serves this purpose. All invocations of native methods and of JNI upcalls are executed remotely via door calls transparently to both Java methods and user-supplied native code. If an error occurs during the native method invocation, for instance if the Jlogin process aborts due to a bus error caused by a user-supplied native library, MVM-2 throws a Java exception in the invoking thread. The exception is unchecked so that existing code is not broken, but still can be caught by applications coded to deal with unexpected failures. Just as in the proxy approach, a custom JNI environment pointer is used to ensure proper forwarding of upcalls back to MVM-2.

Even though in MVM-2 multiple isolates transparently share class and method representations, different isolates from different users are unaffected by the behavior and bindings of each other's native methods. This includes correct handling of pathological cases such as different isolates resolving the same method of the same class to different native methods in different native libraries. To this end two identifiers are introduced: a global (JVM-unique) method id and an isolate-unique library id.

A JVM-unique method id is assigned to each native method at class load time. This id is used in the Jlogin process to bind a native method id to the actual native code that should be called. The binding is performed in Jlogin which guarantees that a particular isolate's bindings do not affect the bindings of any other isolate.

An isolate-unique id is assigned to each loaded library (during `System.loadLibrary()`), identifying a particular library in an isolate's Jlogin for the purposes of method resolution and unloading. This assignment is done at library load time by Jlogin, and the id is later used only between an isolate and its Jlogin.

These two unique identifiers are sufficient to support library loading and unloading as well as method resolution and invocation with the required semantics of giving each isolate an illusion of being the only one in the entire virtual machine.

4.4 Changes to the JVM

Some modifications to the JVM itself were necessary in order to accomplish the goals of the new NCI protocol. MVM was modified to track per-isolate information needed to locate Jlogin for each isolate and to forward load and unload requests to the correct

remote Jlogin process. Similarly, the native method dispatching mechanism of the JVM was modified to forward the native call to the correct Jlogin process.

Library loading modifications included decisions as to whether a native library should be loaded locally within the JVM itself and which libraries to load in the Jlogin processes. Libraries loaded by core classes should not be isolated, and native libraries needed for the actual implementation of NCI must not be isolated. In MVM-2 both of these cases are handled in the same way and are loaded into the virtual machine's process.

On the other hand, user-supplied native libraries must be isolated. In our implementation of the JDK, instances of `ClassLoader` track the currently native loaded libraries for each class. This required modifications to determine which libraries to load remotely and which libraries are "trusted" system code. Finally, the HSVM code that builds run-time representations of methods from class files has been modified to assign JVM-unique method ids to native methods of loaded classes.

4.5 Performance

Much like the overheads of RFA, the performance impact of NCI depends on how often it is used. Isolates that do not invoke user-supplied native methods do not incur any performance penalties. Programs frequently issuing JNI calls may suffer significant overheads. For instance, an empty static downcall through NCI is about 588 times slower than the same call in an unmodified JVM (in absolute terms, the difference is between tens of microseconds versus tens of nanoseconds). Upcalls incur smaller yet significant penalties – a static method upcall is about 45 times slower through NCI than in HSVM. The difference between the NCI's downcall and upcall overheads when compared to HSVM is explained by the fact that in HSVM upcalls are much more expensive than downcalls, and MVM-2 introduces the same overhead to both downcalls and upcalls. However, a vast majority of programs do not issue such a volume of JNI downcalls and/or upcalls related to non-core native libraries to make performance degradation due to this technique noticeable.

5 VIRTUALIZING CORE LIBRARIES

Core native libraries contain the implementations of native methods from core packages, distributed with the JDK. These libraries were typically coded without much thought directed toward safe in-JVM multitasking, let alone multiple users, and contain a substantial amount of static (global) state. To ensure isolation, each isolate needs to have its own copy of such state.

It would seem desirable to handle core native libraries in the same way user-supplied native libraries are executed in MVM-2 – in a separate process. This would provide each isolate with its own copy of core native libraries, and in particular with its own instance of their data segments. We experimented with this approach but soon discovered that certain components of the Java platform make extensive use of their native libraries, and the traffic across the JNI boundary may be heavy. Performance overheads made the use of this technique particularly unattractive for executing native code of core classes associated with the Abstract Window Toolkit (AWT). For example, during the start-up of the Notepad demo application distributed with the JDK there are 2592 downcalls and 156 upcalls. NCI overheads (Sec. 4.4) increase start-up time by an order of magnitude.

Core native libraries are as trusted and robust as the virtual machine itself, and are under full control (i.e., can be modified) of the JVM developers. This observation suggested that only a single instance of each of them may need to be loaded into the virtual machine, provided that the libraries are modified in two ways: their static state is (manually) replicated, and any interaction with the underlying OS is examined to guarantee the absence of inter-isolate interference. This approach still was not satisfactory, since depending on the JDK's implementation the amount of changes required to completely analyze and modify the core native libraries may be substantial. Moreover, the modifications would not extend to the X Window libraries, which are distributed with the operating system and not with the JDK; AWT native code depends on these libraries. GUI-enabled isolates would thus still interfere through the static state of, for example, `libX11`.

The solution we eventually adopted has none of these performance and engineering disadvantages. It is based on a technique that allows for loading multiple instances of the same dynamic library into a single process. In particular, the advantages are (i) memory footprint related to core native libraries does not increase relative to HSVM, and (ii) only minor modifications to the JDK (about 20 lines of code) were required to apply this technique to the AWT subsystem to provide per-isolate static native state. The applicability to Swing/AWT is particularly interesting, as these JDK components generate large amounts of meta-data (loaded classes, compiled method, etc.). Since meta-data is shared in MVM-2, the ability to execute GUI-enabled isolates increases the scope of memory footprint savings and at the same time applies MVM-2's start-up time reduction to interactive applications.

5.1 Basic Technique

The Solaris Operating Environment supports a *runtime linker auditing interface*. Audit libraries can monitor and modify certain operations, such as mapping of shared binary objects into memory and the binding of symbols in a customized way [Sun00b]. Note that we use the term *shared binary object* to refer to a concatenation of relocatable objects (.o files) that provides services that might be bound to a dynamic executable at runtime. A shared binary object may have dependencies on other shared object. In the linker/loader literature the typically used term is *shared object*, which may be confusing when used along the concepts from object-oriented programming.

In order to isolate audit libraries from the symbol binding requirements of the audited applications and shared binary objects, the interface employs its own *link-map* list, with each entry on the list describing a single shared binary object within the process. The symbol search mechanism required to bind shared binary objects for an application traverses this list of link-maps. Thus, the link-map list provides the name-space for process symbol resolution.

The runtime linker itself is also described by a link-map, which is maintained on a different list from that of the application objects. Having the linker reside in its own unique name space prevents any direct binding of the application to services within the linker. The audit interface allows for the creation of an additional link-map list, so that audit libraries are also isolated from the symbol binding requirements of the application.

We have taken advantage of this infrastructure to load multiple copies of a library into a single OS process. Each such library is loaded by the `dlopen()` function [Sun00b] on a new link-map list and at a virtual address different from any other instance of itself in the same process. However, text segments of all instances of the library, regardless of what process's virtual memory they are mapped into, are backed by the same physical memory pages.

5.2 Application to AWT

The technique described above has been applied to MVM-2 and AWT-related libraries. Arguably out of all JDK components with native libraries AWT is the most complex one. For instance, it starts its own threads to handle events and depends on X11 and other related libraries (e.g., Motif), which are in themselves quite complex.

In a nutshell, the approach is to group together the entire set of AWT-related shared libraries as well as the libraries they depend on (such as libX11, etc.) into a unit, called from now on *the AWT context* (or simply *context*). All libraries in the same context share the

same unique link-map list. Due to the name-space isolation provided by separate link-map lists each AWT context can be loaded multiple times within the same OS process without the danger of interference with other contexts. The above is insufficient, however, to provide each isolate with independent AWT capabilities. The major issues that needed to be addressed to make this scheme work are: (i) managing the interface between the virtual machine and multiple AWT contexts, (ii) handling the dependencies of the libraries in AWT contexts on the virtual machine, and (iii) preventing conflicting use of OS resources by multiple AWT contexts.

5.2.1 The JVM-core native libraries interface

Maintaining one AWT context per isolate requires dispatching each invocation of an AWT native method to the AWT context associated with the current isolate.

The JVM (and MVM-2) interface with native code via JNI, and thus the names of core native methods conform to the same stylized naming scheme as an ordinary native library. A simple script based on the standard *nm* utility (listing the symbol tables of shared binary objects) and on *javap* (disassembling classes) is sufficient to generate a list of all such methods, along with their signatures, from the libraries comprising the AWT context. The list is then used to generate *libawt_proxy.so*. At boot time MVM-2 loads a single instance of this library in the main context (i.e., JVM's context). Each function defined there forwards its invocation to an appropriate per-isolate instance of an automatically generated *libawt_context.so*. A new instance of this library is loaded by *libawt_proxy.so*, using `dlopen()`, whenever a new isolate is started. The library is a part of the AWT context, and contains all of the AWT-related JDK and X11 libraries in its list of shared binary object dependencies. Thus loading an instance of *libawt_context.so* loads a new set of instances of these libraries as well, that is, the entire context.

Invocation forwarding does not require any changes to the JDK or the runtime system. Whenever a native method is called, the runtime system finds the required name in *libawt_proxy.so* and calls it (Fig. 2, left side). Only there the actual look-up of the isolate identifier and the associated AWT context takes place. For example here is a method from *libawt_proxy.so*:

```
void Java_java_awt_Color_initIDs(JNIEnv *env, jclass cls) {
    int iid = get_isolate_id();
    context ctx = contexts[iid];
    (*ctx).Java_java_awt_Color_initIDs(env, cls);
}
```

5.2.2 Dependencies on the JVM

AWT native methods invoke Java methods (both application callbacks and services offered by the JDK libraries) exactly as any other native code does: by

invoking the JNI functions of the JNI environment a reference to which is always a first argument to any native method. Although these JNI upcalls are defined in the library that implements the JVM (`libjvm.so`), they are exported to native methods as function pointers. Due to this, JNI functions do not create a dynamic linker dependence from AWT native libraries on `libjvm.so`.

This is however not the case with JNI utility (JNU) functions, which are also defined in `libjvm.so`. Introduced for convenience, each JNU function groups a common sequence of JNI upcalls. An example of a JNU function is invoking a static Java method by its (string) name, class name, and signature. Accomplishing this with JNI requires six upcalls, including appropriate error checking. In contrast to plain JNI upcalls, JNU functions are called directly (i.e., without a function pointer indirection) by AWT native methods, which makes AWT native libraries depend on `libjvm.so`. Such dependencies are undesirable, as loading an instance of `libawt_context.so` would also cause the loading of a new instance of the JVM's libraries along with the new AWT context. In the best case this would only waste memory; in the worst, it can lead to a process crash if a conflict occurs among multiple JVM contexts. For

example, each of them may zero out the heap during initialization.

In MVM-2 such dependencies are prevented by avoiding direct references to the JNU functions. Instead, function pointers are used, much like in the case of the JNI upcalls. In order to achieve this while avoiding the modifications of the original AWT libraries, a new shared library, `libjnu+system.so`, is interposed between the libraries of an AWT context and the JVM. The new library defines all the JNU functions that are used by AWT contexts, and stores the addresses of their original (i.e., defined in the loaded instance of `libjvm.so`) implementations in a vector of addresses `vm_context`. Each such interposing function simply consists of calling the corresponding JNU function of the JVM via the function pointer listed in `vm_context`. The vector is passed to an initialization routine in `libjnu+system.so` when a new instance of `libawt_context.so` is loaded.

5.2.3 System calls

Loading multiple AWT contexts can cause inter-isolate interference because the contexts share the system call interface. For example, while calls to `getpid()` from different contexts are not dangerous, the same cannot be said about `sbrk()`. Each context's `sbrk()` would initialize the amount and initial address of

space allocated for the process's data segment to the *same* value. Subsequent memory allocations through for example `malloc()` invoked from different contexts would return the same address, leading to memory corruption. It is thus vital to neutralize all potentially conflicting uses of system calls.

This issue is solved by extending the technique described in Sec. 5.2.2, where a vector of JNU functions is passed down to an AWT context so that their invocations are properly forwarded back to the virtual machine's context. For this purpose `vm_context` is extended with addresses of (i) system functions (e.g., `sbrk()`), (ii) derivative library functions (e.g., `malloc()`), and (iii) other library functions the use of which must be confined to the virtual machine's context (e.g., `dlopen()`). By itself this does not guarantee that the OS interface is used in an interference-free way, but at least introduces a point of programmable control over potentially dangerous behavior. For example, `malloc()` and `free()` behave as

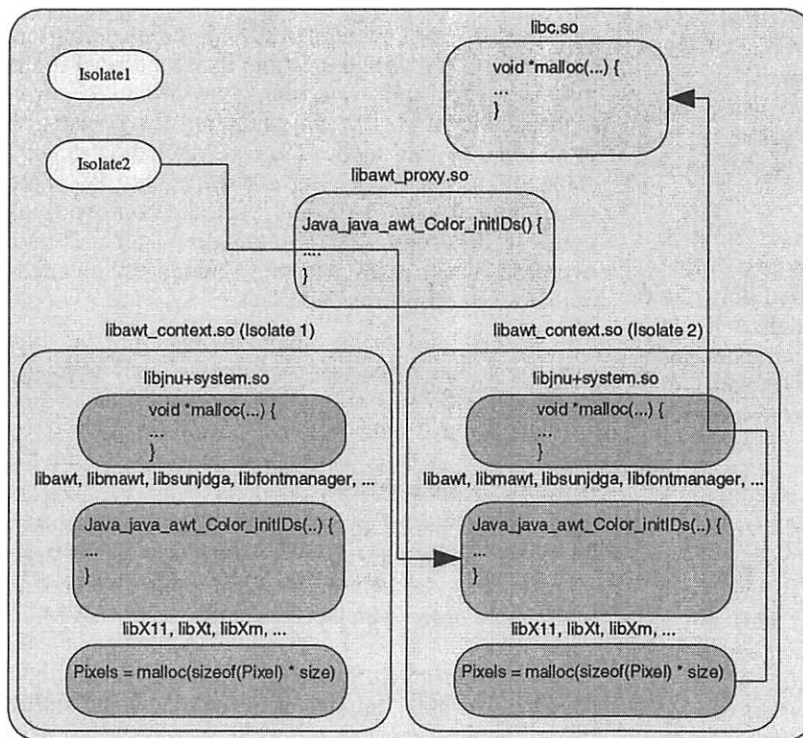


Figure 2. Forwarding of AWT native method invocations to the appropriate AWT context (left) and forwarding system and library calls from an AWT context back to the virtual machine's context.

expected in this scheme, while the usage of signal() system call may have to be modified, for example by injecting transparent chaining of signal handlers and ensuring the chaining will be respected by all contexts. In MVM-2 this has not been necessary for the AWT subsystem. Figure 2 (right side) shows the forwarding of system calls and virtual machine utility functions.

Another way to approach the forwarding of system calls to the main context is to take advantage of the runtime linker auditing interface's ability to intercept requests for loading a new library. This way, a request to load a context-private instance of libc.so could return a library with the necessary redirections, similar to libjnu+system.so. This technique avoids loading an instance of libc.so along with each context. In MVM-2 a new AWT context contains an instance of libc.so, which is never used. As the segments of libc.so are mostly read-only, this does not increase the overall memory footprint, but unnecessarily wastes virtual memory, which may be an issue for 32-bit virtual machines.

5.2.4 JDK modifications

No changes to the virtual machine were needed to enable the replication of the AWT subsystem. Several minor JDK modifications were necessary, though.

In the original AWT native libraries XOpenDisplay() is invoked with a null argument. This means that X11 will call getenv("DISPLAY") to obtain the value of the DISPLAY variable defined in the environment of the virtual machine. MVM-2 obtains this value by contacting the Jlogin process and then passes it to XOpenDisplay() so that each user session has its own value of DISPLAY.

Other modifications concern JNI_OnLoad. Several native libraries that comprise the AWT context define this optional function, which is invoked during the loading of the library to properly initialize it. In MVM-2 the names of all these occurrences of JNI_OnLoad are changed to <libname>_JNI_OnLoad to avoid name clashes. Then, libawt_proxy.so invokes all of these methods in the original order when loading a new instance of libawt_context.so.

5.2.5 Performance and start-up time

Execution time overhead of the presented technique is negligible (we were unable to detect any during our measurements). The reason is that MVM-2 adds only a few machine instructions to JNI downcalls, and even fewer for JNU upcalls and system calls. MVM-2 enjoys the same performance improvements as MVM, since the effort necessary to load classes and compile methods is saved due to meta-data sharing.

In MVM-2 start-up time of GUI applications improves dramatically relative to HSVM. Table 2 summarizes

	First	Second
Notepad	99.3%	28.7%
SwingSet2	99.5%	37.6%

Table 2. Start-up time in MVM-2, relative to HSVM.

the results for two demo programs distributed with the JDK: Notepad and SwingSet2. The first one is generally deemed more representative of desktop applications. The second one creates a large number of Java objects during its start-up, and thus the improvements are relatively smaller, although still very much noticeable by the users. Let us note that we have used the following definition of start-up time for AWT applications: it is the time elapsed between invoking the main method of the application and draining of the AWT event queues. The "first" column reports the time, relative to HSVM, necessary to start up the first instance of the application in MVM-2. The "second" column reports start-up time of the second instance (relative to HSVM's start-up time, which is the same for any instance of the benchmark, as it includes starting a new process, JVM bootstrap, etc.) Start-up times of subsequent (third and later) instances are similar to the start-up time of the second instance. This latter number is more typical of the actual user experience, as most of the start-up time decrease is attributed to having already loaded Swing and AWT classes.

Because of the adopted start-up time definition, bootstrap of the virtual machine is not included in the measurements. If it was, the gains would be even higher, as in MVM the time required for preparation of an isolate to run application code, including running static initializers of bootstrap classes, is only 4% of the time required to boot HSVM. Still, the improvements remove between 62-71% of the start-up time overhead, which translates into shaving off hundreds to thousands of milliseconds.

The barely measurable improvement for the first instance is due to the fact that some classes normally loaded in HSVM during the start-up Notepad and SwingSet2 have already been loaded in MVM-2 by the Mserver isolate manager (Sec. 2.2) before the benchmarks are executed.

Any improvements to the JDK classes will have a direct impact on start-up time in MVM-2, as all static initializers of classes needed by an application are run in the same order in an isolate as they are in HSVM.

5.2.6 Memory footprint

There are several components of the memory footprint of an application executing in MVM-2: (i) Java objects created on the heap, (ii) space occupied by the user-supplied native libraries, (iii) meta-data, such as bytecodes, constant pools, and compiled methods, and (iv) space occupied by core native libraries. The

amount of memory used by the first two is the same in HSVM and in MVM-2.

The size of memory required by native libraries related to AWT in MVM-2 is summarized in Table 3. The size of `libawt_proxy.so` is 576KB, most of which is read-only. This library is not a part of the AWT context and is loaded once by MVM-2.

The total size of the AWT context, loaded for each isolate that uses AWT, is 4920KB, of which 1312KB is attributed to the JDK's native libraries, 32KB to `libawt_context.so` (it includes `libjnu+system.so`), and the rest to X11, Xn, Xt, etc. The read-only portion of the AWT context is 3856KB. It is backed by virtual memory pages shared among contexts. Thus, a new isolate that needs to use Swing/AWT will increase the physical memory footprint by up to 1064KB due to the new AWT context's writable memory. A process executing HSVM would consume the same amount of writable memory for AWT.

Due to separate virtual addresses for each library, an AWT context requires almost 5MB of virtual memory. For single-user desktop systems, where a virtual machine would not typically execute many applications, this does not appear to be problematic with 32-bit JVMs. However, using this technique to run a large number of GUI programs requires 64-bit implementations of the JVM.

Memory footprint reductions due to meta-data sharing can be large, as Figure 3 indicates. The first bar shows the size of meta-data generated during the bootstrap of MVM-2. The next three bars show the amount of meta-data in MVM-2 after starting up one instance of Notepad (second bar), two instances of Notepad (third bar), and three instances of Notepad (fourth bar). The last three bars show meta-data size for the SwingSet2 demo. Meta-data is split up into the following components: (i) *permanent generation* (runtime representation of classes, including bytecodes and constant pools), (ii) *code cache* (size of dynamically compiled methods), (iii) *static state* (size of storage for mutable class components), and (iv) *class mirrors* (data associated with an instance of a class on behalf of an isolate).

Libraries	Read-only	Read-write
JDK (e.g., <code>libmawt</code>)	1208	<u>104</u>
X Window (e.g., X11)	2624	<u>952</u>
<code>libawt_context.so</code>	24	<u>8</u>
<code>libawt_proxy.so</code>	544	32

Table 3. Size (in KB) of components of AWT-related shared libraries in MVM-2. Underlined boldface indicates additional memory required for a new AWT context.

The first two components are the same in HSVM and in MVM-2. However, only in MVM-2 they are (transparently) shared among isolates. Thus, running concurrently three instances of Notepad in three instances of HSVM generate about 13.5MB worth of meta-data memory footprint (4.5MB for each instance). In MVM-2 the footprint is less than 5.1MB. In addition to 4.5MB generated in HSVM to run one instance of Notepad, this includes a slightly growing code cache (since more methods get compiled), constant amount of static state (explained below) and 38KB of class mirrors per isolate executing Notepad. For SwingSet2, which generates more than 7MB of metadata, the memory footprint reduction due to sharing in MVM-2 is even more pronounced.

In MVM-2 access to static fields is indirected through an array indexed by an isolate identifier. The size of these arrays is set to the maximum number of concurrent isolates MVM-2 can execute. This value is a command-line parameter of MVM-2, and in our experiments is set to 32. All such arrays comprise “static state” of an isolate. The static state component remains constant for subsequent executions of the programs if no new classes are loaded. This was the case in our experiments (Figure 3).

It is important to note that the amount of meta-data shared among different applications can be substantial, especially when they use a large JDK component. For example, Notepad run after SwingSet2 generates only

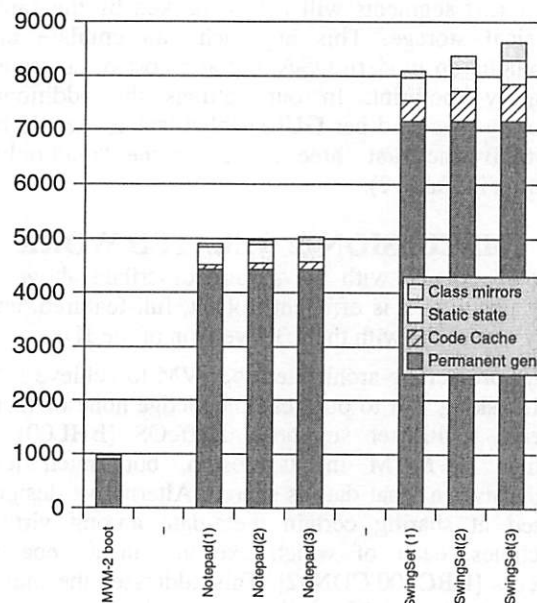


Figure 3. Memory footprint (in KB) related to meta-data of Notepad and SwingSet2.

about 100KB of additional meta data, as both applications rely heavily on Swing and SwingSet2 utilizes a large part of Swing.

Finally, the space occupied by the Jlogin processes must also be taken into account when analyzing memory footprint of MVM-2. The memory-resident image of a single instance of Jlogin that has not loaded any user-supplied native library takes 3.45 MB, 1.55MB of which are shared segments of libraries commonly used by most processes (i.e., libc, libsocket, etc.). Thus, each Jlogin process adds 1.9MB to the overall footprint. 80% of this additional space consumption is caused by the NCI library (Sec. 4), which allocates a large data structure on the process's heap. We are looking into ways of minimizing this overhead.

Interested readers are referred to [Spec98] for selected details of HSVM's performance.

5.3 Portability

The technique described in this section takes advantage of the audit libraries functionality available on the Solaris Operating Environment to map read-only segments of the library's instances to the same physical memory pages. We are not aware of any other OS with a similar functionality and a convenient interface.

On UNIX and win32 systems a library can be renamed and then loaded into a process under a different name. Thus, multiple instances of the same shared library can exist in a single process, but under different names. Their text segments will not be backed by the same physical storage. This approach can emulate our technique on modern OSes, but at a cost of enlarging memory footprint. In our settings the additional memory overhead per GUI-enabled isolate would be 3856KB (the first three entries in the "read-only" column in Table 2).

6 DISCUSSION & RELATED WORK

Our experience with the system described above is very positive: it is efficient, robust, full-featured, and fully compliant with the 1.3.1 version of the JDK.

Other projects re-architected the JVM to achieve safe multitasking, but to our best knowledge none of them offered multi-user support. KaffeOS [BHL00] is similar to MVM in its design, but much less aggressive in what data is shared. Alternative designs aimed at sharing certain meta-data among virtual machines, each of which executes in a separate process [DBC+00,CDN02]. This addresses the multi-user issue, but scales much more poorly than a single multitasking virtual machine.

In contrast to MVM-2, which simultaneously maps multiple "logical" JVMs onto a single OS process, another approach to the affinity between the virtual machine and the OS process it runs in is exploited in SAP's Process Attachable Virtual Machine (PAVM) [KKL+02]. When a user session starts, an instance of PAVM is created. It has a private session memory block, which stores the complete computational state of the session, as well as run-time data structures such as thread stacks and heap area needed by the session. This organization allows PAVM instances to be mapped into any work process and then unmapped, made persistent, and eventually mapped back into the same or another work process, since the session state is completely encapsulated in the session memory. Thread scheduling is cooperative, which is beneficial for maximizing batch processing throughput. Special care has been taken to properly handle bindings to external resources by sessions.

The .NET platform [Micr02] defines *application domains*, similar to the notion of isolates. Instances of System.AppDomain are virtual processes isolating applications from one another. Multiple application domains can exist in a single OS process. There is no multi-user support – all collocated application domains execute on behalf of the same user. Moreover, unlike MVM-2's isolates, .NET's application domains cannot safely use arbitrary native code.

Several approaches could be applied to make native code memory-safe. However, the techniques such as augmenting native code with safety-enforcing software checks [WLA+93], statically analyzing it and proving it safe [NL96], or designing a low-level, statically typed target language to compile native code to [MCG+99] are unsuitable for our purposes. The major problem is these mechanisms add memory safety only, and do not address the issue of conflicting use of OS resources. Moreover, they either introduce non-negligible performance overheads, require source code for re-compilation, or are not general enough for complex native code. An approach applicable to MVM-2 to remove IPC costs introduced by NCI is Protected Shared Libraries [BTC97]. A protected shared library may have, within the process that loaded it, its own protection domain. Having such functionality in an OS would elegantly address the problem of collocating the JVM and untrusted native libraries. Our solutions, based on peer native processes to isolate untrusted code and the ability to load the same set of libraries multiple times with the same process for trusted code, are dictated by a pragmatic consideration: using the features available on a mainstream operating system.

An interesting design alternative is to re-implement an OS in the Java programming language. This would subsume certain functionality that currently requires peer native processes (e.g., instances of Jlogin) in MVM-2. For example, file access could be done entirely in the safe language. This can lead to savings and optimizations, e.g., file access permissions would be checked only once, via Java security. Hardware protection would not be required, so a part of the overhead currently due to process switching would not be incurred.

Several OSes have been implemented entirely or almost entirely in a safe language. Earlier systems, such as Cedar [SZB+86], were typically single-user. An exception is Project Oberon, which associated user identities with executing programs. However, such support was minimal, as the premise of that system was that the Oberon server “operates in a harmonious environment” ([WG92], p. 324). A more recent example is the SPIN operating system [BSP+95], implemented in Modula-3. JavaOS™ [SM99] was a first attempt to offer the complete OS functionality entirely in the Java programming language. However, the system was single-user and consisted of a single protection domain. A more complete and ambitious environment is the JX operating system [GFW+02], also written in the Java programming language.

Comparing MVM-2 with JX illustrates several points. JX does not rely at all on hardware protection. This implies that a arbitrary user-supplied native library can jeopardize the whole kernel and all applications, and is thus disallowed. In MVM-2 native code executes in a separate process, but a virtual machine bug can corrupt or abort all the current computations. The reliability of the virtual machine can improve and eventually become as robust as a kernel of a commodity OS, while a decision to entirely eliminate memory protection from the OS prevents native code from ever being run (unless the need for native code does not exist at all). Designs such as JX or JavaOS are promising but only for applications coded entirely in the Java programming language. Purity has significant advantages, though – for example, if the AWT subsystem was implemented without any native code in our base JDK, as it is done by the Remote Abstract Window Toolkit [IBM98], the issue of virtualizing core native libraries would be much less important in the development of MVM-2.

7 CONCLUSIONS

We have implemented extensions to a multitasking virtual machine that allow it to safely host isolated computations from various users. MVM-2 builds on the safety of the Java programming language to collocate multiple programs within a single-address space. The identities of users are preserved with

respect to operations that need them. The virtual machine is transparently shared among computations, with improved resource utilization. Any standard API of the Java platform including file access, native code, and the graphical subsystem can be used by any computation from any user with an illusion of having the JVM all to itself.

The performance penalties for using user-supplied native code and file access operations are proportional to how often the application uses those features; in the case of remote file access only certain operations incur overhead, while reading and writing to files do not suffer any performance impact. Using the graphical subsystem does not result in any performance overhead and at the same time significantly lowers both the start-up time and the memory footprint.

In this paper we have not described any resource control mechanisms of MVM-2. This is partially due to space constraints and partially because this work is still in progress [CHS+02]. Certainly such facilities are needed before we could call MVM-2 a multi-user platform akin to an OS for isolates.

Our approach takes to the extreme the concept of the virtual machine executing on top of a commodity OS. Auxiliary processes are needed to provide multiple operating system resource and user contexts, but no modifications are required to the OS itself and at the same time no feature of Java platform is missing or compromised. We view MVM-2 as a step towards gradual blending of the functionality and implementations of virtual machines and operating systems.

Acknowledgments. The authors are grateful to Ciaran Bryce, Dave Dice, Mick Jordan, Doug Lea, Miles Sabin, Glenn Skinner, Alex Snoeren, Pete Soper, Pat Tullmann, Jan Vitek, and Mario Wolczko for their comments, suggestions and help. Special thanks are due to Rod Evans for explaining details of linkers and loaders and to Fred Oliver for sharing his insights about start-up time measurements.

Trademarks. Sun, Sun Microsystems, Inc., Java, JVM, Enterprise JavaBeans, HotSpot, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. SPARC and UltraSPARC are a trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

8 REFERENCES

[BHL00] Back, G., Hsieh, W., and Lepreau, J. *Processes in KaffeOS: Isolation, Resource*

- Management, and Sharing in Java*. 4th OSDI, San Diego, CA, 2000.
- [BTC97] Banerji, A., Tracey, J., Cohn, D. *Protected Shared Libraries: A New Approach to Modularity and Sharing*. USENIX Annual Technical Conference, Anaheim, CA, January 1997.
- [BSP+95] Bernshad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. *Extensibility, Safety and Performance in the SPIN Operating System*. 15th ACM SOSP, Copper Mountain, CO, December 1995.
- [BV99] Bryce, C. and Vitek, J. *The JavaSeal Mobile Agent Kernel*. 3rd International Symposium on Mobile Agents, Palm Springs, CA, October 1999.
- [CD01] Czajkowski, G., and Daynes, L. *Multitasking without Compromise: A Virtual Machine Evolution*. ACM OOPSLA'01, Tampa, FL.
- [CDN02] Czajkowski, G., Daynes, L., and Nystrom, N. *Code Sharing among Virtual Machines*. ECOOP'02, June 2002, Malaga, Spain.
- [CHS+02] Czajkowski, G., Hahn, S., Skinner, G., and Soper, P. *Resource Consumption Interfaces for Java Application Programming - A Proposal*. ECOOP'02 Workshop on Resource Management for Safe Languages, Malaga, Spain, June 2002.
- [DBC+00] Dillenberger, W., Bordwekar, R., Clark, C., Durand, D., Emmes, D., Gohda, O., Howard, S., Oliver, M., Samuel, F., and St. John, R. *Building a Java virtual machine for server applications: The JVM on OS/390*. IBM Systems Journal, Vol. 39, No 1, 2000.
- [GFW+02] Golm, M., Felser, M., Wawersich, C., Kleinoder, J. *The JX Operating System*. The USENIX Annual Technical Conference, Monterey, CA, June 2002.
- [GJS+00] Gosling, J., Joy, B., Steele, G. and Bracha, G. *The Java Language Specification*. 2nd Edition. Addison-Wesley, 2000.
- [HCC+98] Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D. and von Eicken, T. *Implementing Multiple Protection Domains in Java*. USENIX Annual Conference, New Orleans, LA, June 1998.
- [IBM98] IBM Corp. Remote AWT For Java. alphaworks.ibm.com/tech/remotewtforjava.
- [JCP01] Java Community Process. JSR-121: Application Isolation API Specification. <http://jcp.org/jsr/detail/121.jsp>.
- [KKL+02] Kuck, N., Kuck, H., Lott, E., Rohland, C., and Schmidt, O. *SAP VM Container: Using Process Attachable Virtual Machines to Provide Isolation and Scalability for Large Servers*. Work-in-Progress Session, Java Virtual Machine Research and Technology Symposium, San Francisco, August 2002.
- [Lian99] Liang, S. *The Java Native Interface*. Addison-Wesley, June 1999.
- [MM01] Mauro, J., and McDougall, R. *Solaris Internals - Core Kernel Architecture*. Prentice Hall, 2001.
- [Micr02] Microsoft Corp. *.NET Web Page*. <http://www.microsoft.com/net>. 2002.
- [MCG+99] Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., and Zdancewic, S. *TALx86: A Realistic Typed Assembly Language*. ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA, May 1999.
- [NL96] Necula, G., and Lee, P. *Safe Kernel Extensions without Runtime Checking*. 2nd Symposium on Operating Systems Design and Implementation, Seattle, WA 1996.
- [PCD+02] Palacz, K., Czajkowski, G., Daynes, L., and Vitek, J. *Incommunicado: Efficient Communication for Isolates*. ACM OOPSLA'02, Seattle, WA, November 2002.
- [SM99] T. Saulpaugh and C. Mirho. *Inside the JavaOS Operating System*. Addison-Wesley 1999.
- [SZB+86] Swinehart, D., Zellweger, P., Beach, R., and Hangmann, R. *A Structural View of the Cedar Programming Environment*. ACM Transactions on Computer Languages and Systems, Vol. 8. No. 4, October 1986.
- [SLN99] Schmidt, B., Lam, M., and Northcutt, D. *The Interactive Performance of SLIM: a Stateless, Thin-Client Architecture*. 17th ACM SOSP, Kiawah Island, SC, 1999.
- [Spec98] Standard Performance Evaluation Corporation. <http://www.spec.org>.
- [Stev90] Stevens, R., *UNIX Network Programming*. Prentice Hall, 1990.
- [Sun00a] Sun Microsystems, Inc. *Java HotSpot™ Technology*. <http://java.sun.com/products/hotspot>.
- [Sun00b] Sun Microsystems, Inc. *Linker and Libraries Guide*. <http://docs.sun.com>.
- [Sun02] Sun Microsystems, Inc. *New I/O APIs*. <http://java.sun.com/j2se/1.4/docs/guide/nio>.
- [WLA+93] Wahbe, R., Lucco, S., Anderson, T., and Graham, S. *Efficient Software Fault Isolation*. 14th ACM SOSP, Asheville, NC, December 1993.
- [WG92] Wirth, N. and Gutknecht, J. *Project Oberon*. Addison-Wesley, 1992.

A Logic File System

Yoann Padioleau and Olivier Ridoux

IRISA / University of Rennes 1

Campus universitaire de Beaulieu

35042 RENNES cedex, FRANCE

{padiolea,ridoux}@irisa.fr, <http://www.irisa.fr/lande>

Abstract

On the one hand, hierarchical organizations are rigid in the sense that there is only one path to each document. On the other hand, keyword-based search is flexible because many sets of keywords may lead to the same document, but it lacks a navigation mechanism. We present the new paradigm of a *logic file system*, which integrates navigation and classification, and the possibility of expressive queries. This paradigm associates logical descriptions to files, and logical deduction serves as a basis for navigation and querying; *paths are formulas*. A key notion is the *extension* of a logical formula: i.e., the set of all files whose description satisfies the formula. The root directory is determined by formula *true*, and sub-directories of a directory are determined by formulas whose extension strictly intersects the directory extension. This gives a logical ground for considering navigation as computing *relevant hints* to help refining a query. A prototype implementation demonstrates encouraging performances.

1 Introduction

Computer users can access a very large variety and number of digital documents. Managing so many documents is a difficult task. An important difficulty is to find quickly one desired document among many files; this is the information retrieval problem. Information retrieval techniques cannot be thought independently of organization methods; they form *paradigms*. The most well-known information retrieval/organization paradigms are the *hierarchical paradigm*, the *boolean paradigm*, and the *relational paradigm*. This introduction continues by presenting the advantages and disadvantages of these paradigms. The Logic File System is an original combination of these three paradigms.

1.1 Hierarchical organizations

Traditional file systems belong to the *hierarchical paradigm*, as well as Web portals such as *Yahoo!*, and many e-mail managers. Information is organized by first creating a hierarchy of *concepts*, which induces a tree-like structure, and then putting *objects* in this tree, which is the classification process. Information is searched by going up and down in this tree; this is the navigation process. In a file system, concepts are *directories*, and objects are *files*.

The advantages of the hierarchical paradigm are:

- the system proposes navigation hints to the user, e.g., directory names. If the user does not remember or does not know the exact classification of an object, then the computer will help him.
- the system proposes relevant hints only. By first proposing global concepts and then sub-concepts, the system helps in finding an object step-by-step.

The disadvantages are:

- since there is a single path to a file, one must know it exactly. This requires either luck or erudition. Gopal and Mander say [8] "Beyond a certain scale limit, people cannot remember locations for explicit path names, after so many years, it is still amusing to see even experienced UNIX system administrators spend time trying `/usr/lib` or was it `/usr/local/lib`, maybe `/opt/local/etc/lib` or `/opt/unsupported/lib`. There are of course many search tools available, but organizing large file systems is still too hard". Our objective is to take into account this *human factor*; users of a file system tend to look for files that they do not know where they are. So, users should not commit themselves to expressing definite paths.
- one can put an object in only one place, which means one cannot associate several independent (i.e., not ordered by the subconcept relation) concepts to one object.

However, one often wants to associate several independent concepts to an object: e.g., price and weight.

- another disadvantage is that the logic of the query language is limited to conjunctive formulas (e.g., a/b for traditional file systems), though disjunction is a clean formal way of expressing non-definite paths. Disjunction can be simulated using shell tricks, such as in

```
ls -r */(keyword1|keyword2) / ,  
but cd */(keyword1|keyword2)/ means nothing. Negation is still more difficult to simulate. To express non-definite paths, a user would like to do just cd !keyword1, or cd keyword1|keyword2.
```

1.2 Boolean organizations

Web search engines such as Google belong to the *boolean paradigm*. Information (in the Web case, HTML files) is organized by describing information elements with keywords. One finds an information by specifying the keywords that its description must contain. It can be extended by boolean formulas, such as

$$keyword_1 \vee keyword_2 \wedge keyword_3.$$

The advantages of the boolean paradigm are:

- As opposed to the hierarchical paradigm, it is easy to describe an information element by a conjunction of concepts. It is flexible because of the many boolean formulas that are equivalent. E.g., queries $man \wedge ps$ and $ps \wedge man$ are equivalent, as opposed to a hierarchical file systems where man/ps and ps/man are not equivalent at all.
- The query language is expressive; it permits conjunction, disjunction and negation.

The disadvantages are:

- As opposed to the hierarchical paradigm, the system does not answer a query with relevant keywords; instead, it returns a list of all information elements whose description is partially matched by the query. So, the exact keywords that permit to select an information element must be guessed by the user; the system does not help.
- Since the system answers with a list of information elements the result of a query can be long, which can mean useless, because a user cannot/does not want to go through the entire list to find the desired file. That means that the user needs to find further keywords without help.

So, one would like that a boolean system proposes relevant keywords that will refine the search; one wants a navigation capability with a boolean query capability.

1.3 Relational organizations

Databases belong to the *relational paradigm*. Data is organized by first creating tables, according to a pre-established *schema*, and then by filling in those tables with items. One searches an object by using a complex language based on relational algebra.

The big advantage of databases is the expressiveness of their query language. E.g., one can associate valued attributes to items, and then make complex queries such as

```
select ... where size > 45.
```

The difficulty is that this expressiveness comes at a price. Databases are more complex systems than file systems, and are less easy to learn (a UNIX-shell course: 1 hour, an SQL course: 40 hours). This is because there are many concepts to learn, such as selection and projection. Moreover, as for boolean organizations, it does not propose navigation, which makes it difficult to find an information without knowing the schema of the database.

1.4 A new paradigm

All paradigms described in this introduction have their advantages and drawbacks. The main contribution of our work is to propose a new organization which makes it possible to combine navigation, querying, ease of use, and expressiveness, and which can be implemented reasonably efficiently. As opposed to previous works (see Section 6 for related works), the combination of querying and navigating is unrestricted. In particular, one can navigate in the result of any query. This organization is an instantiation of a theoretical approach [3, 4] that generalizes file paths to almost arbitrary logic formulas. Information systems based on this approach are called *Logic Information Systems* (LIS for short). Once an organization is chosen, the question remains of what will be its place in a system architecture. The proposed organization is implemented as a file system. This makes it usable by all sorts of applications, e.g., shells, editors, compilers, multimedia players. The file system will be called *Logic File System* (LIFS for short, and to avoid confusion with the Log-structured File System [16]).

The article is organized as follows. We first present the principles and usage of a logic file system in Section 2. Then we expose an implementation scheme, with its data structures and algorithms in Section 3. We present additional features and extensions to the basic scheme in Section 4. Section 5 describes an actual implementation

and the results of experiments. Section 6 presents related works. Finally, we present future search directions and conclusions in Sections 7 and 8.

2 Principles of a logic file system

We first describe a file system based on the boolean query paradigm and then we extend it to add the navigation capability. In terms of Gopal and Mander example (see Section 1.1), the boolean file system allows to do `ls lib` to get every library files. With navigation, the answer is not formed of all library files, but of all keywords that can be used to make the query more precise: in their example, `usr`, `opt`, ... Then, the user can do `ls lib/usr`, and the answer will be all files that are fully identified by keywords `lib` and `usr` (in fact, the files of UNIX directory `/usr/lib`), plus keyword `local`, because it helps identifying files that are in `usr/lib/local` and not in `usr/lib`. Keyword `opt` is not listed because it is not relevant to `lib/usr`. In this framework, `/` commutes, so that `usr/lib` and `lib/usr` are equivalent.

The boolean file system plus the navigation capability forms the core of the logic file system. We present the semantics of shell commands in the resulting file system. Then we explain how this new paradigm affects the security model. We describe a file system in terms of shell commands, because they are more user-oriented. Only when entering into deeper details in Section 3.3, do we consider actual file system operations, like `lookup` and `readdir`.

2.1 A boolean file system

As we have chosen a standard file system interface we have to deal with files and directories. Boolean properties will play the role of directories. To handle boolean queries in a file system, we associate a conjunction of properties of interest (i.e., a directory) to every file. This is done by first registering property names with command `mkdir`, e.g., `mkdir man`; `mkdir latex`; ... At this stage, this indicates that `man` and `latex` are subdirectories of `/`, but it also indicates that `man/latex` and `latex/man` are paths to potential subdirectories. Second, command `cd` sets the working directory (variable `PWD` in some shells) to a desired list of property names. Finally, commands like `touch` and `cat` will create

		properties							
	o	name(o)	a0	a1	a2	a3	a4	a5	a6
ls = { f1, f2, f3, f5, f7, }	1	f1	x	x		x	x	x	
	2	f2	x		x	x	x	x	
	3	f3				x	x	x	
	4	f4				x			
	5	f5				x	x	x	
	6	f6						x	x
	7	f7	x	x		x	x	x	

objects

PWD = a3 / a5

Figure 1: a boolean file system

files associated with the properties named in the PWD, e.g., `cd man/latex/french`; `touch myfile`. Note that `latex` needs not be a subconcept of `man`, nor `french` be a subconcept of `latex`, as in a hierarchical file system. The slash (`/`) must be read as a conjunction, hence it is commutative.

Boolean logic defines a language, i.e., its formulas, and an entailment relation that is usually written \models . $f_1 \models f_2$ means that if f_1 is true, then f_2 must be true too. For instance, $a \wedge b \models a$ holds in boolean logic. In the current prototype, formulas must be presented in conjunctive normal form: e.g., a , $\neg a$, $a_1 \vee a_2 \vee a_3$, or $(a_1 \vee a_2) \wedge a_3 \wedge (\neg a_4)$. These formulas are written a , $!a$, $a1|a2|a3$, and $(a1|a2) \& (a3) \& (!a4)$ in the concrete syntax. The entailment relation is that of usual boolean logic.

Let \mathcal{O} be the set of all the files in the file system, $d(o)$ be the description of a file o (in our case, $d(o)$ is a conjunction of properties), $name(o)$ be the name of file o , and $c(o)$ be the content of file o . The state of a boolean organization is well-represented by a $name \times property$ matrix (see Figure 1 for an illustration). The answer to `ls` in a PWD p is $\{name(o) \mid o \in \mathcal{O}, d(o) \models p\}$. This set is called the *extension* of formula p . Note the risk of name clashes in extensions; several o 's with different descriptions, but same name, may clash in an extension. This risk will disappear in the final design of Section 2.2. The root directory, or `/`, is equivalent to formula *true*. So, doing `ls` at the root will list the names of all the files in the system, because anything entails *true*.

The PWD evolves incrementally. Given a property x , doing `cd x` in a PWD p , changes the PWD into $p \wedge x$. A logic path must always be composed with the PWD, tak-

ing into account relative and absolute paths, and special names like ". . ". E.g., doing `cd /y` sets PWD to `y`.

As files evolve, their descriptions evolve too. So, one needs a way to update the description of a file. In hierarchical file systems, the place where a file is located is the current description of this file, and when one wants to modify its description, one just changes its location using command `mv`. This works in the same way in a boolean file system. Concretely, if one executes `mv p1/f1 p2/f2`, the effect is to change the name of `f1` into `f2`, and to change its description, deleting attributes of `p1` and adding those of `p2`. E.g., assume file `f` is created with description `man^latex^french`, then command `cd /man/ps; mv f ../pdf` results in the new description `man^pdf^latex` for file `f`.

Executing `rmdir x` removes a property name (i.e., a virtual directory). It first checks that `x` is a simple atom (neither a disjunction, a conjunction, nor a negation), and that this property is empty, i.e., $\{o \mid o \in \mathcal{O}, d(o) \models x\} = \emptyset$. Finally, `rm` proceeds as in a hierarchical file system.

At this stage, a boolean file system has all the advantages, but also the drawbacks, of a boolean organization. In the following section, we add navigation to the boolean file system. This defines the full *logic file system*.

2.2 Boolean file system, plus navigation

The answer to a boolean query can be very large (typically, `cd /; ls` would list the extension of the root, i.e., every file), so one would like to add a navigation capability to the boolean file system. So doing, the system will answer a query with *relevant* properties that refine the search. Those properties will be presented as sub-directories. By "relevant" we mean that if in a PWD `p` all files have keyword `a1`, some have the keyword `a2`, and none have the keyword `a3`, then neither `a1` nor `a3` is relevant to distinguish between the files, but `a2` is relevant. Command `ls` lists sub-directories, but also files that cannot be distinguished any further by relevant properties. In a logic file system, the files located in a PWD `p` are the files whose description satisfies `p`, but does not satisfy any of its subdirectories. In the preceding example, if one of the files had just the properties mentioned in `p` and `a1`, and nothing more (i.e., not `a2`), then it would be listed.

More formally, let \mathcal{F} be the set of all property names, let $ext(f) = \{o \in \mathcal{O} \mid d(o) \models f\}$ (the extension of `f`), then

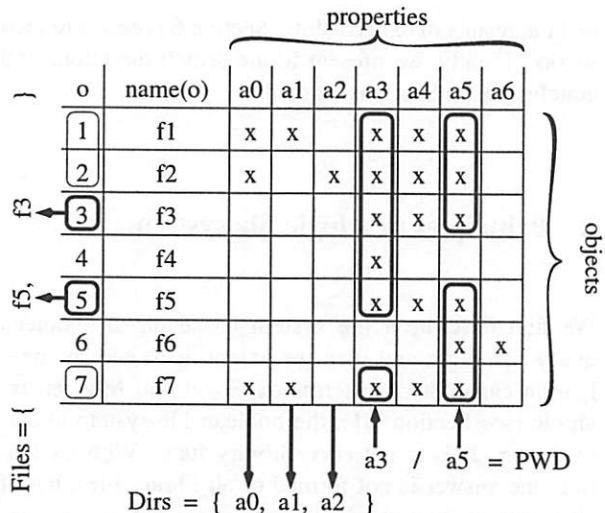


Figure 2: querying and navigating

the answer to `ls` in a PWD `p` is divided in two parts, the sub-directories *Dirs*, and the files *Files*, such that

$$Dirs = \{f \in \mathcal{F} \mid \emptyset \subset ext(f \wedge p) \subset ext(p)\}$$

and

$$Files =$$

$$\{o \in \mathcal{O} \mid d(o) \models p, \neg \exists f \in Dirs \mid d(o) \models f \wedge p\}$$

(see Figure 2 for an illustration). Sub-directories in *Dirs* are also called *increments*.

In hierarchical file systems, several files may have exactly the same name, provided that they are not located in the same directory; otherwise, the user would not be able to disambiguate which file he wants to manipulate. In the boolean file system, the same problem arises, so one must ensure that if two files have the same name, they must not have exactly the same description. If this condition is kept true, a path always exists where only one of these two files is listed, and so where there is no ambiguity. Navigation always finds this path. For instance, users Alice and Bob may both have a file called `foo`; one file `foo` will have attribute `alice`, and the other one will have `bob`. Alice in her homedir `/home/alice` will see her own `foo`, and Bob in his homedir `/home/bob` will see his own `foo`. A file `foo` may also have attributes `alice` and `bob`; in this case, it is visible by both users.

2.3 Navigation in a taxonomy

With this scheme, the number of file names listed by a command `ls` is reduced, but there may be too many increments anyway. For instance, at the root directory, the system would list nearly all the property names. The principle of navigation is obviously to propose concepts

that refine the search, but also to propose the most general such concepts. E.g., assuming computer science documents, one would like to classify the keywords so that the system first proposes the main fields of computer science, e.g., *algorithms*, *databases*, *operating-systems*, and then the subfields, e.g., *UNIX*, *Linux*, *Windows*.

So, one needs some means for stating that an atomic property is more general than others. To this aim, we use the `mkdir` command to create a hierarchy of concepts: a *taxonomy*. So, to say that a property a_1 is more specific than a_0 , one simply executes `mkdir a0; cd a0; mkdir a1`, as for files. If an object x has property a_1 , then it must also have property a_0 . In logical terms, this means that $d(x) \models a_1$ must entail $d(x) \models a_0$. So, the extension of a_0 must contain the extension of a_1 . This is achieved by considering the taxonomic relations as axioms, e.g., $a_1 \models a_0$. Note that the taxonomic relation is a *directed acyclic graph* (a DAG). E.g., doing

```
cd /OperatingSystem/TradeMark
mkdir Unix
```

makes Unix a subconcept of OperatingSystem (axiom $\text{Unix} \models \text{OperatingSystem}$), and of TradeMark (axiom $\text{Unix} \models \text{TradeMark}$; UNIX is a registered trademark of The Open Group). Now, if a user does `ls /OperatingSystem` all files with property Unix will also satisfy this query.

Command `ls` is refined accordingly, by proposing the most general increments. So, the answer to `ls` in a PWD p becomes

$\text{Dirs} = \max_{\models}(\{f \in \mathcal{F} \mid \emptyset \subset \text{ext}(f \wedge p) \subset \text{ext}(p)\})$, where

$\max_{\models}(\mathcal{F}) = \{f \in \mathcal{F} \mid \neg \exists f' \in \mathcal{F}, f' \neq f, f \models f'\}$, and Files does not change (see Figure 3).

2.4 A security model

One of our goals in designing LISFS was to be as much compatible as possible with existing file systems. So, there is no reason to change the way file permissions are handled beyond what is implied by the new navigation paradigm. This is quite easy to do for files, because their nature does not change with LISFS, but we had to design a new security model for directories, because their nature changes a lot with LISFS.

Trying to mimic the way hierarchical file systems handle directory permissions introduces the following question: what are the permission/ownership of a conjunction of property names such as man/ps ? Under hierarchical

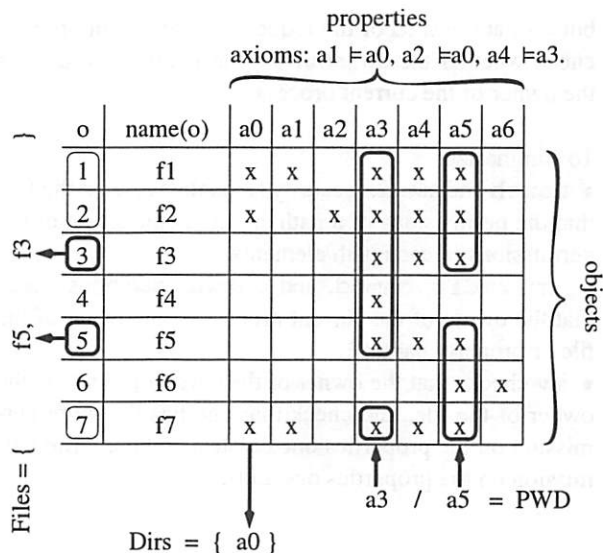


Figure 3: querying and navigating in a taxonomy

file systems, it is the permission/ownership of the last directory on the path (ps in the example) that gives the permission/ownership of the whole path. This is sensible since creating a file in ps does not modify the content of man . However, in the logic file system, the order of property names in a path is immaterial. So, the permissions of man/ps should be a conjunction of the permissions of man and ps to make it certain that, if the owner of man has so decided, nobody can create a new object in man or man/ps . This way of handling directory permissions makes commands that create objects (e.g., `touch` or `mkdir`) safe.

LISFS permits that a user “sees” a file, without specifying all its keywords; it suffices to specify enough keywords so that the file is designated unambiguously. This is like seeing any distant subdirectory in a hierarchical file system. This has strange side effects on security; one can see a file one does not own, from a directory where one has the write permission. This means that, under the traditional security semantics, one could either delete this file, even without the write permission on all the properties describing this file, or add a property to the description of a file (with the `mv` command). So, we refine the security model by allowing only the owner of a file to delete or change the property list of a file. The same kind of difficulty arises in hierarchical file systems with the directory `/tmp`. Every user can create files in `/tmp`, which implies the write permission for all on `/tmp`. However, one does not want that a user deletes the files of another user. The “sticky-bit” solves this difficulty. In a directory having this bit, the system does not look only at the permissions of the directory,

but also at the uid of the requesting process in order to check whether the owner of the file is the same user as the owner of the current process.

To summarize,

- touch and mkdir security semantics rely on the fact that the permissions of a path are the conjunction of the permissions of each path elements.
- rm, rmdir, chmod, and chown operations check that the owner of the current process is the owner of the file or property name;
- mv checks that the owner of the current process is the owner of the file, and check that one has the write permission on the properties one deletes, and the write permission on the properties one adds.

3 Algorithms and data structures

We will first describe the LS algorithm that computes the answer to ls as it is the most original LISFS operation. Indeed, it computes the sets *Files* and *Dirs* for getting local files and sub-directories (see definition of *Dirs* and *Files* in Section 2.2), whereas the usual operation performed by ls is merely to read a directory file. Moreover, this operation dictates the choice for the data structures that we will present just after. Then we will give an overview of the concrete implementation of a few representatives LISFS operations.

3.1 The LS algorithm

Each property name in \mathcal{F} is represented by an *internal keyword identifier* f_i , and each object in \mathcal{O} is represented by an *internal object identifier* o_i . To represent the description $d(o)$ of an object, a table `object->keywords` indexed by internal object identifiers contains lists of internal keyword identifiers (indeed, in this prototype, the description of an object is a conjunction of properties). This corresponds to rows of the *name* \times *property* matrix. A formula is represented by a list of a union value, which is either an internal keyword identifier, or the tag `Or` associated with a list of internal keyword identifiers (the operands of the disjunction), or the tag `Not` associated with an internal keyword identifier. The list will play the role of the conjunction. The axioms of the taxonomy are represented as a DAG of internal keyword identifiers implemented as a table `keyword->children` and a table `keyword->parents`. We call it the *taxonomy DAG*.

Adding an axiom $x \models a \wedge b \wedge c$ means attaching x as a child of the internal keyword identifiers a , b and c . In this data structure, the top node represents the most general property, i.e., *true* because anything implies *true*.

Extensions are not computed with table `object->keywords`; instead, they are pre-computed in the inverted table `keyword->objects`. This uses standard indexing techniques, and this corresponds to the columns of the *name* \times *property* matrix. The extension of $a \wedge b$ is computed by intersecting a and b extensions, i.e., $ext(a \wedge b) = ext(a) \cap ext(b)$. Similarly, $ext(a \vee b) = ext(a) \cup ext(b)$, and $ext(a \wedge \neg b) = ext(a) \setminus ext(b)$. The underlying assumption for using an inverted table is that the number of attributes per path and file description is small wrt. the number of files. This is confirmed by experiments, and by analysis. Indeed, most of the attributes are computed automatically by functions that do not depend on the number of files (see *transducers* in Section 4).

In fact, the table `keyword->objects` does not contain for every property only the objects described by this property, but also the objects that are described by some sub-property of this property. This amounts to precompute the extensions of all the atomic properties. Then, adding a file with property x requires to update (using table `keyword->parents`) the `keyword->objects` entry of x and of all its ancestors in the taxonomy DAG. This is costly, but we think that the main operation to optimize is consultation rather than creation. So, we prefer to transfer the hard-work to creation commands. Moreover, creation does not require an immediate update; its response time can be good anyway if it leaves the computation to a background task.

So, table `keyword->objects` returns extensions at a constant cost. The PWD formula is usually small, because it is often the trace of a navigation by a human being. So, to compute extension (PWD) costs only a few accesses ($\|PWD\|$) to table `keyword->objects`. Moreover, extensions can be compressed by using an interval representation (where lists of consecutive elements are represented by their *min* and *max*), and specialized set operations can be used. This is especially useful for the most general properties which contain in their extensions nearly all the objects.

The algorithm first locates the internal keyword identifiers that are present in PWD, and computes the extension of PWD. Then it searches the taxonomy tree downwards from the root node while the extensions of traversed nodes contain the PWD extension. The highest internal keyword identifiers whose extension does not con-

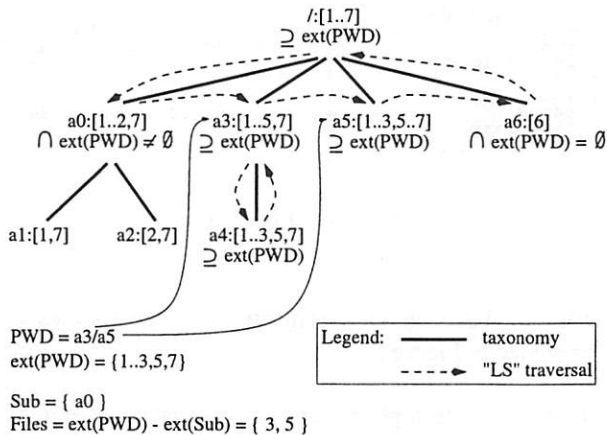


Figure 4: the final algorithm

tain the PWD extension, and has a non empty intersection with it, are the maximal increments to list in *Dirs* (see pseudo-code below). This avoids having to go through all the formulas. The extensions of all sub-properties of a property that does not intersect with the PWD are not considered, because if `keyword->objects[x]` does not intersect with PWD then no descendant of *x* will intersect since `keyword->objects[x]` contains the “in-lined” extensions of its children. Similarly, extensions of sub-properties of a property that strictly intersects with the PWD are not considered either, because those sub-properties will also intersect with the PWD but are not maximal increments.

```

extension(f) =
  let set =
    forall x in the list f
      compute intersection of
        either x is a single keyword
          then keyword->objects[x]
        either x is a OR with elements xs
          then
            forall y in xs
              compute the union
                of keyword->objects[y]
        either x is a Not y
          then do nothing
  in ext := set
    forall x in the list f
      if x is a Not y then
        ext := minus(ext,
          keyword->objects[y])
      otherwise do nothing
  return ext

LS(f) =
  let ois = extension(f)
  let fis =
    collect keywords fi

```

```

from the taxonomy DAG
using a depth first search
starting from the top node
using keyword->children
until
  card = cardinality(
    inter(extension(fi),
      ois))
  0 < card < cardinality(ois)
in Dirs = fis
Files = minus(ois,
  union_extension(dirs))

```

Figure 4 illustrates this algorithm. The context is the same as in Figure 3. Every node of the taxonomy is represented by its name and extension (e.g., `/:[1..7]` for the root node). The thick lines represent the taxonomy. The two thin arrows point to the atomic formulas of the current PWD. The dashed arrows represent the actual traversal done by the algorithm for answering command `ls`. For each traversed node that has subnodes, the algorithm compares the extension of the node with the extension of the PWD; the result of this comparison is written under the node description (e.g., `⊇ ext(PWD)` for the root node). Only when the result is `⊇ ext(PWD)` does the algorithm enter the subnodes. Sub-directories are those that do not contain PWD but have a non-empty intersection with it (e.g., *a*₀). *a*₆ does not count as a sub-directory because it does not intersect PWD. Files in the extension of PWD but not in a sub-directories are returned (i.e., {3, 5}).

The complexity of computing *Files* and *Dirs* is essentially the complexity of the product of a *name* × *property* matrix by a vector of properties (the PWD), plus the product of the transposed matrix by a vector of names (the *Files*). The first product computes the *Files* as of the boolean file system (see Section 2.1), and the second product computes the *Dirs*. Assume there are *P* properties and *N* names. Both products cost *P* × *N* operations. However, both the matrix and the vector of properties are sparse. Indeed, the number of attributes of a file seldom depends on the number of files. So, one can assume it is constant. This is confirmed by experiments; e.g., the coding of a set of man pages as a logic file system costs about 45 attributes per file whatever the number of files. Let us call *p* the number of properties per file, *p* ≪ *P*. Moreover, the vector of properties (the PWD) has usually less non-zero elements than *p*. Let us call it *π*, *π* < *p*. Then the first product costs only *π* × *N*. It results in a vector of names with *n* non-zero elements, *n* ≪ *N*. The second product costs *P* × *n*. Finally, since the scalar operations are boolean operations, they have good absorbing properties (e.g., *x* ∧ *false* = *false* whatever is *x*). This allows

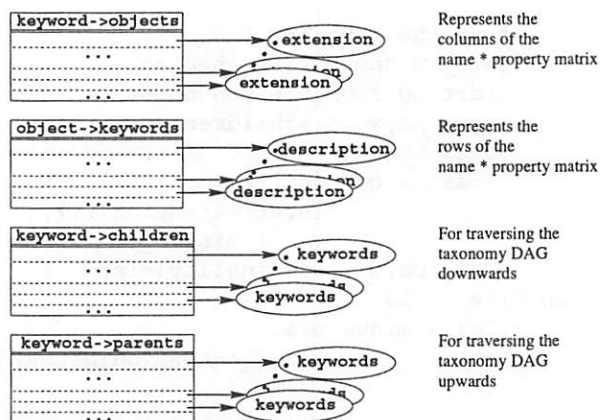


Figure 5: the main tables

to short-circuit the computation, without computing every internal products. The essence of algorithm LS is to perform these matrix-vector products, taking advantage of sparseness and taxonomy axioms, and short-circuiting products as soon as possible. In summary, assuming the number of properties per name does not depend on the number of names, the worst-case complexity of algorithm LS is in $\mathcal{O}(N)$. It is much less in practice because of the caching of answers and the short-circuits in boolean operations. Note that the total number of files, N , must be considered with respect to the current PWD. Assume a machine with a million files; this number only affects operations at the root of the file system. As soon as one moves to subdirectories with smaller extensions, N decreases. Section 5 presents the measured cost of LISFS operations on a prototype.

3.2 Data structures

The internal data structures use internal object and keyword identifiers, instead of plain names. This is more space and CPU efficient. In a traditional file system, these internal identifiers are the inode number of a file (or of a directory). In our specification, o 's are such internal object identifiers. The meta-data consists mainly in tables `keyword->children` (which represents the taxonomy DAG) and `keyword->objects` (which represents the extension table) used by the LS algorithm. Moreover, in order to preserve the consistency of extensions, command `touch` updates the entries in `keyword->objects` for all the ancestors of the internal keyword identifiers in the current PWD. This is done recursively using a table `keyword->parents`. Similarly, command `rm file` updates the extensions of all internal keyword identifiers used in the description of `file`. This uses a table `object->keywords` which

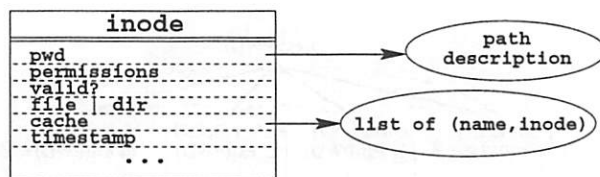


Figure 6: the inode

was called $d(o)$ in our specification. All these tables are presented in Figure 5.

In order to return plain names in `ls` answers, we introduce two tables to get the name of a file or property given its internal identifier: `keyword->keywordname` and `object->filename`. As we allow to do `cd x` even if x is not a subdirectory of the current directory, we need to know to what internal identifier x corresponds to, so we add a table `keywordname->keyword`. In order to check, when adding a file, that there is no other file with the same name and same description, we also introduce a table `filename->objects`.

In the UNIX context, a typical user manipulates files with names, whereas the operating system uses an internal identifier: the inode number. The inode structure is presented in Figure 6. Traditional file systems store on disk an `inode_table` (indexed by the inode number) where each entry (aka. the inode) contains the control information to manage a file or a directory, such as its mode, permissions, data block addresses, etc.

Algorithm LS starts from the PWD formula, which it knows only as an inode. So, we add a `pwd` field to directory inodes, which contains a list of a union type. Each union value contains either the internal property identifier of a property, or the tag `Or` with a list of internal property identifiers (the operands of the disjunction), or the tag `Not` with an internal property identifier.

Moreover, answers to `ls` are cached. Directory inodes contain the addresses of blocks that store the result computed by `ls`, which mimics the contents of a directory under a hierarchical file system. As this result must be recomputed every time someone modifies the contents of a LISFS, inodes contain a local timestamp indicating the time the current result was computed. LISFS maintains a global timestamp that is incremented every time someone adds or deletes a file or an atomic property. Operation `ls` compares the global timestamp with the local one, to decide if the increments must be recomputed.

All this means that every time new increments are computed, fresh inodes are allocated, and their local time-

stamp is set to 0 (to force the next call to `ls` to recompute increments). When increments are recomputed, LISFS will fill in storage blocks with the result computed by the LS algorithm, and will adjust accordingly the block addresses in the inode.

3.3 Concrete operations

We switch now from shell operations to file system operations to enter in more details. We use Linux VFS terminology (*Virtual File System* [10]). To save space, we enter in the details of only a few LISFS operations. Note that these details are often direct consequences of the data structures, so that details of other operations can be inferred for a large part.

For fault-tolerance, we use journaling for implementing *transactions*. Every time several tables must be updated in an atomic way, a transaction is started, which logs the updates. In case of failure, the next reboot will redo (or undo) the log (see [16] on journaling and transaction). For example, renaming a keyword needs to update table `keyword->keywordname` and its inverse table `keywordname->keyword`. This must be done in a transaction to ensure that the two tables are coherent.

File system operations can be divided into three groups: global operations (aka. superblock operations), directory operations (aka. inode operations), and file operations. Global operations deal with the management of the file system: (un)mounting, and statistics. Directory operations deal with navigation/querying, and creation/deletion of files/properties. Finally, the file operations deal with the file contents. Note that in hierarchical file system, operation `readdir` is considered as a file operation, though we consider it as an inode operation. This is because in these systems to `readdir` is actually to read the content of a *directory* file, though in our case it implies a real computation. Directory operations are original wrt. hierarchical file systems, whereas file and global operations are mostly similar to their counterparts in hierarchical file systems.

3.3.1 Global operations

Operation `read_super` is called via the user program `mount`. It locates the different tables on the disk, it stores this information in the superblock structure for further use, and it fills in the first entry in `inode_table` that corresponds to the root inode. The `pwd` field of this

entry is set to a list of one element containing a single internal identifier of the top node of the taxonomy DAG which corresponds to property *true*. The local timestamp is set to 0, and the global timestamp is set to 1.

Operations `put_super` (shell command `umount`), `write_super` and `stat_fs` (shell command `df`) are like their counterparts in hierarchical file systems.

3.3.2 Inode operations

Operation `readdir` is called via the user program `ls`. It takes as a parameter an inode, and returns a list of pairs containing the plain name of a file or property and the corresponding inode number. If the global timestamp is equal to the local timestamp of the inode, then the list of pairs is read at the block addresses in the inode. Otherwise, an actual computation must be started: algorithm LS (described in Section 3.1) is applied to the `pwd` field of the parameter inode. The result is a pair `Dirs` and `Files` which contain respectively a list of internal keyword identifiers and a list of internal object identifiers. Then, a new buffer is allocated, and for each `oi` in `Files` the pair `(object->filename[oi], oi)` is stored in the buffer. For each `fi` in `Dirs`, a fresh inode number `ino` is allocated, and the pair `(keyword->keywordname, ino)` is also stored in the buffer. Then, for each `ino`, the `pwd` field of `inode_table[ino]` is filled in with the conjunction of `fi` and the current PWD, and its local timestamp is set to 0. Finally, previous blocks used by the contents of this directory are freed, new blocks are allocated and filled with the contents of the current buffer, the local timestamp is set to the value of the global timestamp, and the list of pairs contained in the buffer is returned.

Operation `lookup` can be called via the command `cd keyword`. It takes as a parameter an inode (the current directory) and a string `s`, which is the plain name of a file or of a formula, and it returns the corresponding inode, or an error condition. Operation `readdir` is called to look if the string is in the list of pairs, in which case the corresponding inode is returned. Otherwise, the string must correspond either to a property not listed as an increment of the current directory, or to a complex formula. If the string has the form of a single name, then `keywordname->keyword[s]` gives the corresponding internal keyword identifier `fi`. If this entry is empty, an error code is returned, otherwise a fresh inode `ino` is allocated, and the `pwd` field of `inode_table[ino]` is filled in with the conjunction of `fi` and of the PWD of the current directory, and inode `ino` is returned. If

the string has the form of a disjunction $x|y|z|\dots$, the atomic property names are translated into their internal keyword identifier x_i, y_i, z_i, \dots . If there is a property name without a corresponding internal keyword identifier, then an error code is returned. Otherwise, a fresh inode is allocated as above, putting this time in the `pwd` field a conjunction of the PWD of the current directory with the disjunction (with the `Or` tag) of the internal keyword identifier x_i, y_i, z_i, \dots . If the string has the form of a negation $!x$, then the operation is similar, but putting this time in the `pwd` field the tag `Not`.

Operation `create` can be called via command `touch`. It takes as a parameter an inode (the current directory), a string `s`, and returns the inode corresponding to the object just created. The current directory must correspond to a conjunction, `fis`, of atomic properties (as we have restricted the description of object). This is checked first. Then a similar check is done for the string `s`; it must not contain connectives `|`, `&` or `!`. Finally, another object `y` with the same name and description must not exist. This is checked by looking for an object `y` in table `filename->objects[s]`, and comparing `fis` with the corresponding entry in `object->keywords[y]`. If one of those checks fails, then an error code is returned, otherwise a transaction is started for updating several tables in an atomic way. Then, a fresh internal object identifier `o` is allocated. It is added in `filename->objects[s]`, `object->filename[o]` is set to `s`, `object->keywords[o]` is set to `fis`, and for each keyword `f` in `fis`, `o` is added to `keyword->objects[f]` and recursively in all the ancestor nodes of `f` using table `keyword->parents`. Then, the transaction is ended, and `o` is returned.

Operation `mkdir` behaves the same way as `create`, but this time for atomic properties. It allocates a fresh keyword internal identifier, `fi`. Then, it adds new entries in tables `keywordname->keyword` and `keyword->keywordname`, and it sets to empty `keyword->children[fi]` and `keyword->objects[fi]`, and for each atomic property, `pi`, of the PWD, it adds `fi` in `keyword->children[pi]` and `pi` in `keyword->parents[fi]`.

Operation `unlink` undoes what has been done in `create`, checking that the current user is the owner of the file to be deleted.

Operation `rmdir` behaves the same way as `unlink`, but for atomic properties.

Finally, operations `notify_change` and `read_inode` manage the `inode_table`, associating the appropriate permissions to an inode, using the security semantic described in section 2.4.

3.3.3 File operations

Operations `lseek`, `read`, `write`, `open`, `release`, `truncate` are standard. For example, operation `read` takes as a parameter an inode, and a buffer to be filled in. It first gets the inode number `oi`, looks in `inode_table[oi]` to get the block addresses of the contents of the file and fills in appropriately the buffer passed in parameter.

4 Extensions

Section 2 exposed the core of LISFS. In reality this is only a framework, in which more features can be introduced. We present in this section a part of the features that have actually been introduced in the prototype LISFS. A more complete account, particularly on ACL-like security and variants of navigation, is given in a research report [14].

LISFS includes the possibility of valued attributes. For instance, one can create a directory `author:minsky`, or `size:45`. Since related properties can be grouped in a taxonomy DAG, one can gather all properties of the form `size:x` as sub-concepts of the property `size`. This increases the readability of `ls`, which will propose first the coarse categories (`size`, `author`, ...) and then the finer sub-categories, that is the valued attributes (`size:1`, `size:2`, ...). LISFS supports operations on integer attributes, e.g., `cd size:>45` and string attributes, e.g., `cd auteur=~ m.?in.+y.*`. We simulate such a query by constructing a disjunction of all the existing properties that satisfy the condition, e.g., `cd (size:46|size:72|...)`. This is done in the `lookup` operation.

As many properties can be automatically inferred from the file contents, we designed *transducers* which are functions that extract attributes and values from file contents (as in the Semantic File System [7]). If attributes are new, they are created on-the-fly. In our prototype, transducers are defined for all the system attributes of a file, e.g., its size and last modification time, but also for its extension, e.g., `ext:c` is extracted from

foo.c. For the sake of experimentation, we have also defined a transducer for MP3 music files, which extracts the genre, author and year from the meta-data encoded in the file. This permits requests like `cd genre:Disco/year:1980`. We could also easily define other transducers to support more file types, but we prefer to offer the user a simple interface to define his own transducers.

Conceptually, the description of a file is split in two parts: the *extrinsic* part, made of properties assigned by the user, and the *intrinsic* part, made of properties inferred by transducers. As the content changes, the intrinsic part changes too. This is done in operations `release` and `notify_change`. Intrinsic attributes are not updated by the `read` or `write` operations. Indeed, calling the transducers is costly, and we prefer to update the intrinsic attributes only when the user closes a file, that is when doing `release`. To update the intrinsic part, each transducer is called in turn, with the contents, name and system attributes of the file as arguments.

5 Experimental results

The current prototype of LISFS is implemented as a user level file system, using PerlFS. We use Berkeley DB [13] to manage the different meta-data (implemented as Btrees). The transactional module of Berkeley DB provides the necessary tools for preventing the possible corruption of meta-data by a crash. Finally, we use the underlying file system (EXT2) to manage the contents of files and tables.

Since there is no similar file system to compare with, a part of our experimentations aims at evaluating the overhead of LISFS with respect to a similar technology file system that implements the standard semantics. Since the LISFS prototype is built upon EXT2 and PerlFS, we evaluate the overhead with respect to these file systems. Another part of our experimentations aims at evaluating the performance of LISFS for tasks it has been designed for, like information retrieval. In this case, it is compared with a user-level application that performs the same task, like command `find`. We ran several experiments to determine the overhead of using LISFS, both in speed and disk space. The platform for all experiments was a Linux box running kernel 2.4, with a 2Ghz Pentium 4, 750Mb RAM, and a 40 Gb IDE disk.

The first experiment evaluates the disk space overhead used by the meta-data of LISFS (see Table 1). The ex-

	Andrew	MP3	Man
NbFiles	860	633	11502
FSize	10 Mb	1772 Mb	246 Mb
TSize	2 Mb	3.1 Mb	43.3 Mb
AvNbAttr	26/23	36/20	21/24
NbAttr	1686	3730	43442
AvFSize	11.6 Kb	2799 Kb	21.4 Kb
SpOverH%	20 %	0.17 %	17.6 %
SpOverH/F	2.3 Kb	4.9 Kb	3.7 Kb
SpOverH/A	1.2 Kb	0.84 Kb	1 Kb
SpOverH/FA	47 bytes	87 bytes	84 bytes

NbFiles = Number of files, FSize = Total size of files, TSize = Total size of LISFS tables, AvNbAttr = Average number of file attributes (intrinsic/extrinsic), NbAttr = Total number of attributes, AvFSize = Average file size, SpOverH% = Space overhead (per cent), SpOverH/F = Average space overhead per file, SpOverH/A = Average space overhead per attribute, SpOverH/FA = Average space overhead per attribute of file.

Table 1: Results of Disk Space Benchmark

periment is run for a set of 633 MP3 music files, a set of 860 program files obtained by ten copies of the Andrew file system benchmark [9], and a set of 11502 man pages. The Andrew files are described by intrinsic attributes valued by the names of functions declared in them (as produced by command CTAGS). The MP3 files are described by intrinsic attributes valued by MP3-specific meta-data such as genre and artist. The man pages are described by the words of their description line. All have extrinsic attributes for ACL-like security and for representing their position in a user-defined hierarchy, plus other intrinsic system attributes for size, last modification time, etc. SpOverH/F measures the average cost of file descriptions (i.e., rows of the *name* \times *property* matrix). SpOverH/A measures the average cost of each attribute (i.e., extensions, or columns of the matrix). SpOverH/FA measures the average cost of each individual attribute of each file (i.e. each non-empty position in the matrix).

In the second experiment, we ran the modified Andrew benchmark, first on the native file system (EXT2), then on a hierarchical file system implemented via PerlFS, then on LISFS where transducers were turned off then on (see Table 2). The Andrew benchmark has 5 phases: *Mkdir* constructs a directory hierarchy, *Copy* copies files, *Scan* recursively traverses the whole hierarchy, *Read* reads every byte of every file, and finally *Make* compiles the files. Note row *Read* where LISFS without transducer is faster than PerlFS because PerlFS goes into empty directories that LISFS avoids because they are not relevant. We also ran our own benchmarks that consists in creating

	Ext2	PerIFS	LISFS (transducer off)	LISFS (transducer on)
<i>Mkdir</i>	0.217s	0.986s	1.823s	3.703s
<i>Copy</i>	1.359s	5.943s	13.212s	46.296s
<i>Scan</i>	2.506s	5.141s	5.348s	6.638s
<i>Read</i>	3.548s	11.510s	11.119s	12.333s
<i>Make</i>	16.896s	28.384s	36.182s	46.260s
Total	24.526s	51.964s	67.684s	115.230s
MP3	2min28s	4min30s	5min	5min30s
Man	22min	29min	44min	85min

Table 2: Results of CPU Benchmark

the 633 music files, and creating the 11502 man pages. In both cases, creating also means indexing through the use of adhoc transducers.

We now compare the speed of search-like activities using UNIX *find* and *apropos*, and LISFS lookup. With the Andrew files *find*ing function *GXfind* takes 2.899 seconds, but *look*ing it up takes 0.081 seconds. Similarly, with the MP3 files *find*ing *disco* music takes 3.292 seconds, but *look*ing it up is immediate. Doing *ls change* with the man pages takes 1.370 seconds, and returns 110 increments. Doing *apropos change* takes 0.145 seconds, and returns 288 items. In other words, the increments reveal the organization of the items. Doing *ls change/directory* with the man pages takes 0.026 seconds, and returns 4 entries. Doing *apropos change | grep directory* takes 0.030 seconds, and returns the same 4 entries. We have also tested the speed of *ls* in directories of various sizes in the music files context. In directory *artist*, *ls* computes 155 increments in 0.405 seconds. In directory *size*, it computes 629 increments in 1.058 seconds, and finally it computes 28 increments in directory *genre* in 0.220 seconds. Note that in any case, only the first *ls* in a given directory takes time, as for further *ls* LISFS uses its cache, and answers immediately.

Figure 7 plots the creation times of the 633 MP3 files and 11502 man pages. It shows an almost constant creation time until 10000 files, and then a deterioration. We believe that a better representation of extensions will push this point to a greater value.

In summary, the space overhead is tolerable, and it could still be decreased by using better marshalling techniques. Operations *lookup* and *readdir* do not show a great performance penalty, especially considering the work they perform. Moreover, they compare positively with search tools like *find*. On the opposite, opera-

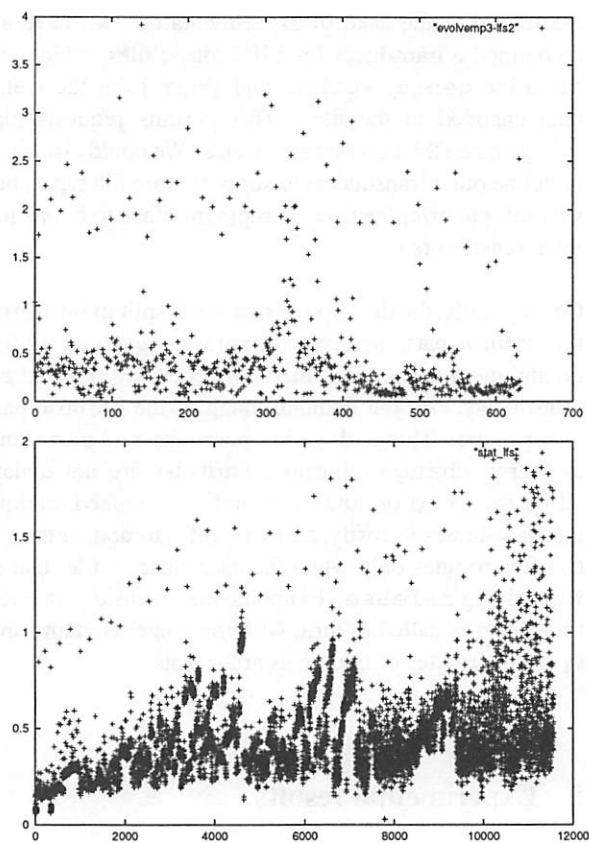


Figure 7: creation times (sec)

tion *create* suffers from a large performance penalty, because the extensions of the taxonomy DAG must be checked and updated. This is visible in the *Copy* row of Table 2. It can be cured by performing most of the *create* operation lazily, in the background during idle time. Indeed, the result returned by *create* does not depend on the updating of internal tables. We need also study alternative representations of the taxonomy DAG.

6 Related work

The Semantic File System (SFS [7]) was the first file system to combine querying and navigation. It remained compatible with the file system interface by using *virtual directories*. Some attributes were extracted from the contents of the files by *transducers*. This allowed users to express queries such as `cd ext:c/size:15`. However, users could not assign their own attributes to a file (i.e., all these attributes were *intrinsic*, see Section 4). More importantly, querying and navigation were two separated operation modes; one could not navigate

in a virtual directory that resulted from a query. The Hierarchy and Contents proposal [8] solved this problem in a way that leads to inconsistencies. Directories that represent answers to queries are no longer virtual; they are real directories in which one can navigate, and even write. However, one can write something which is inconsistent with the query that created the directory. Another file system that combines querying and navigation is the Be file system (BeFS [6]). Following the observation that hierarchical file systems fail to describe a file by a conjunction of independent concepts, BeFS allows the user to manually assign attributes to a file. But this extension is not compatible with a standard UNIX interface (as opposed to SFS and LISFS where one can use classical shells). So, one can either use a shell or browser and navigate, or use the new interface and do querying, but not both. Finally, the Nebula file system [1] allows a user to assign multiple attributes to a file and formulate query at the shell level. One can also create kinds of directories called *views*, which are just names assigned to queries, as for databases. Views can be organized in a hierarchy, where subviews refine parent's views with another query (restricting the set of objects). This allows to cache frequently used queries. One can both navigate following the subviews links, and query the file system, but as for SFS, one cannot navigate in the result of a query.

The principal contribution of our work is a seamless combination of querying and navigation, under the file system interface. The key features are to combine intrinsic and extrinsic descriptions, and to permit navigation in query results by computing relevant subdirectories.

This idea of combining querying and navigation via increments is not new. In the literature, increments are also called *co-occurrence lists*, *term suggestions*, *relevant informations*, *significant keywords*, ... In [11], a corpus of keywords is extracted from man pages, and via formal concept analysis [5], a lattice of keywords is computed to permit a user to find man pages by keywords, and getting as a result other keywords considered relevant (as increments in Section 2.2). Queries are limited to conjunctions of keywords, and keywords cannot be ordered which mean we get the disadvantages mentioned in Section 2.3. In [15], a similar approach is applied to bibliographic information retrieval. The querying mechanism of these applications is completely encompassed by LISFS; the answers of LISFS are the answers of formal concept analysis. In fact, the domain of information retrieval is aware of the need for integrating querying and navigation (e.g., see [2, 12]). However, the proposals in this domain remain at the application level, and are very often combined with visual interface issues.

All those systems are limited to conjunctive queries, and are more like front-ends over another information system for allowing to combine querying and navigating, which means that they do not handle updating in their interface. Our contribution is to offer all these services, querying, navigating, and updating, at the system level, so that many kinds of application can be built on it.

7 Future directions

There is much room for performance improvement in the prototype LISFS. E.g., operations `create` and `readdir` are too expensive. Tricks such as grouping of commands (as in X Window), amortization, lazy structures or use of idle time will certainly improve the performance of `create`. Finally, in place of a global timestamp, locating more precisely what is dirty could lead to less cache miss. Another future work is to make a “complete” logic file system, allowing arbitrary formulas in object descriptions as well as in queries. This requires to incorporate an automatic theorem prover for representing the \models relation in LISFS. The goal here is to permit an open-ended range of file description styles. Even if some logics are undecidable (e.g. predicate logic) or unpracticable (e.g., propositional logic), there are many useful and practicable logics that could be used as a file description language: e.g., a logic of types for program components, or a logic of intervals for expressing dates. Another direction is to overcome the difference between directories and files. We would like to navigate in files in the same way as in directories. E.g., one would like to navigate inside a BibTeX file, or inside a program source file. Then, a user could do `cd !comment & security` to get all the parts of a source file that are not comments and that talk about security.

8 Conclusions

We have presented a new file system paradigm which allows to freely combine high-level file description and querying using logic formulas, navigation, and updating. This is called a *Logic File System*. The integration of querying and navigation goes beyond previous proposals; coherence is kept when writing in virtual directories, and navigation and querying can be freely intermingled. Such a file system gives at a system level services that are useful in many applications. A key technical aspect is to develop data structures and algorithms that permit

to implement a prototype LISFS with encouraging performances. Experiments show that though the prototype LISFS is slower than a more classical one, it passes usual benchmarks with reasonable performances: create-intensive benchmarks show a bad performance ratio for LISFS, but ls-intensive benchmarks show almost no penalty. Consider also that what operation `create` actually does is on-line indexing. Note also that the current implementation of LISFS is very *soft*, a user-level program based on PerlFS, and it could be improved by using more effective techniques. We believe that all this confirms the validity of LISFS.

9 Acknowledgments

We thank the anonymous referees and our sheperd, Prof. Darrell Long, for their thoughtful remarks. We also wish to acknowledge the collaboration with Sébastien Ferré for elaborating the theory of Logic Information System.

10 Availability

A prototype LISFS and more information on this project can be down-loaded at the following URL:

<http://www.irisa.fr/LIS>

References

- [1] C.M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti. A File System for Information Management. In *ISMM Int. Conf. Intelligent Information Management Systems*, 1994.
- [2] P. Bruno, V. Ehrenberg, and L.E. Holmquist. Starzoom - an interactive visual interface to a semantic database. In *ACM Intelligent User Interfaces (IUI) '99*. ACM Press, 1999.
- [3] S. Ferré and O. Ridoux. A file system based on concept analysis. In Y. Sagiv, editor, *Int. Conf. Rules and Objects in Databases*, LNCS 1861, pages 1033–1047. Springer, 2000.
- [4] S. Ferré and O. Ridoux. Searching for objects and properties with logical concept analysis. In *Int. Conf. Conceptual Structures*, LNCS 2120. Springer, 2001.
- [5] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.
- [6] D. Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers, 1999.
- [7] D.K. Gifford, P. Jouvelot, M.A. Sheldon, and J.W. O'Toole Jr. Semantic file systems. In *13th ACM Symp. Operating Systems Principles*, pages 16–25. ACM SIGOPS, 1991.
- [8] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *3rd ACM Symp. Operating Systems Design and Implementation*, pages 265–278, 1999.
- [9] H.J. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, N. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.
- [10] S.R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer*, pages 238–247, 1986.
- [11] C. Lindig. Concept-based component retrieval. In *IJCAI95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs*, 1995.
- [12] R. Miller, O. Tsatalos, and J. Williams. Integrating hierarchical navigation and querying: A user customizable solution, 1995.
- [13] M.A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–192, 1999.
- [14] Y. Padioleau and O. Ridoux. A logic file system. Research Report RR-4656, INRIA, 2002.
- [15] G.S. Pedersen. A browser for bibliographic information retrieval based on an application of lattice theory. In *ACM-SIGIR'93*, pages 270–279, 1993.
- [16] M. Rosenblum and J.K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

Application-specific Delta-encoding via Resemblance Detection

Fred Douglass

IBM T. J. Watson Research Center
Hawthorne, NY 10532

douglass@acm.org

Arun Iyengar

IBM T. J. Watson Research Center
Hawthorne, NY 10532

aruni@us.ibm.com

Abstract

Many objects, such as files, electronic messages, and web pages, contain overlapping content. Numerous past research projects have observed that one can compress one object relative to another one by computing the differences between the two, but these *delta-encoding* systems have almost invariably required knowledge of a specific relationship between them—most commonly, two versions using the same name at different points in time. We consider cases in which this relationship is determined dynamically, by efficiently determining when a sufficient resemblance exists between two objects in a relatively large collection. We look at specific examples of this technique, namely web pages, email, and files in a file system, and evaluate the potential data reduction and the factors that influence this reduction. We find that delta-encoding using this resemblance detection technique can improve on simple compression by up to a factor of two, depending on workload, and that a small fraction of objects can potentially account for a large portion of these savings.

1 Introduction

Delta-encoding is the act of compressing a data object, such as a file or web page, relative to another object [1, 13]. Usually there is a *temporal* relationship between the two objects: the latter object exists, and when it is subsequently modified, the changes can be represented in a small fraction of the size of the entire object. There is often also a *naming* relationship between the objects, since a modified file can have the same name as the original copy. In these cases, identifying the base version against which to compute a delta is straightforward.

Delta-encoding is particularly attractive for situations where information is being updated across a network with limited bandwidth. For example, web sites are often replicated both for higher performance and availability. The bandwidth between the replicas can be limited. Another example would be replicated mail systems. Electronic mail systems often allow clients to replicate copies of mail messages locally. Clients may be connected to the network via phone lines with limited bandwidth. For an email client connected to a mail server

via a slow link, techniques which minimize bandwidth required for updates are highly desirable. However, in each of these environments, it is not always possible to identify an appropriate base version to take advantage of delta-encoding.

Our work therefore addresses a domain in which there are very many objects with arbitrary overlap among different pairs of objects, and the relationships between these pairs are not known *a priori*. If one can identify which pairs are suitable candidates, delta-encoding can reduce the size of one relative to another, thereby reducing storage or transmission costs in exchange for computation. We consider several application domains for this technique, which we refer to as *delta-encoding via resemblance detection*, or DERD: web traffic, email, and files in a file system.

We defer additional discussion of our research until after a more detailed discussion of delta-encoding and resemblance detection, which appears in the following subsection. After that, the next section describes the framework of our analysis in greater detail, including the metrics we consider. Section 3 presents the various datasets we used. Section 4 describes the experiments, and Section 5 provides the results of these experiments. Section 6 discusses the resource usage issues that would arise in a practical implementation of DERD. Section 7 surveys related work, and Section 8 summarizes and describes possible future work.

1.1 Background

It is difficult to describe our approach without providing a general overview of both delta-encoding and resemblance detection. We cover enough of each of these areas here to set the stage for combining the two, then return to a more comprehensive comparison with related work toward the end of the paper.

Deltas are useful for reducing resource requirements, and existing applications of deltas generally fall into two categories: storage and networking. For storage, when one already stores a base version of a file, subsequent versions can be represented by changes. This lowers storage demands within file systems (the Revision Control System (RCS) [25] is a longstanding example of this), backup-restore systems [1], and similar environ-

ments.

Over a network, transmitting data that are already known to the recipient can be avoided. The most common approach in this case is to work from a common base version known to the sender and recipient, compute the delta, and transmit it. This technique has been applied to web traffic [16], IP-level network communication [24], and other domains. An extension to the traditional web delta-encoding approach is to select the base version by finding similar, rather than identical, URLs [7].

What if one wishes to find a similar file based on *content* rather than *name*, among a large collection of files? Manber devised a method for extracting **features** of files based on their contents, in order to find files with overlapping content efficiently [14]. He computed hashes of overlapping sequences of bytes (also known as “shingles”), then looked for how many of these hashes were shared by different files. Manber indicated that clustering similar files for improved compression would be an application of this technique. Broder used a similar approach but used a deterministic sampling of the hash values to dramatically reduce the amount of data needed for each file [5, 6]. With his approach, a subset of features of a file is used to represent the file, and if two files share many of those features in common, there is a high probability of significant content in common as well. A common use for this technique is to suppress near-duplicates in search engine results [6], and variations of the technique have been used in link-level duplicate suppression [24] and file systems [8, 17, 20].

Because the shingling technique has seen so much use in the systems community of late, we refrain from providing a detailed description of it. Briefly, it uses Rabin fingerprints [21] to compute a hash of consecutive bytes; the key properties of Rabin fingerprints are that they are efficient to compute over a sliding window, and they are uniformly distributed over all possible values. Thus, Broder’s approach of selecting the N fingerprints with the smallest values effectively selects N “random” features in a deterministic fashion, and two documents with many features in common overall would hopefully have many of these N features in common.

1.2 Goals

As Manber suggested, one can use the features of documents to identify when files overlap and then delta-encode pairs of overlapping files to save space or bandwidth. One goal of this work was to assess whether this technique is generally applicable, and if not, to identify some specific instances in which it is applicable. A second goal was to evaluate a number of the parameters used in this process, such as:

- the size of a shingle,

- the amount of overlap among features necessary to get a sufficiently small delta,
- the number of files with similar overlap necessary to get close to the “best” delta,
- selection of delta-encoding algorithms and parameters to those algorithms,
- whether delta-encoding the contents of specially formatted files such as Zip files in an application-specific method is beneficial,
- and other metrics.

1.3 Summary of Results

We have found that the benefits of application-specific deltas vary depending on the mix of content types. For example, HTML and email messages display a great deal of redundancy across large datasets, resulting in deltas that are significantly smaller than simply compressing the data, while mail attachments are often dominated by non-textual data that do not lend themselves to the technique. A few large files can contribute much of the total savings if they are particularly amenable to delta-encoding. Application-specific techniques, such as delta-encoding an unzipped version of a zip or gzip file and then zipping the result, can significantly improve results for a particular file, but unless an entire dataset consists of such files, overall results improve by just a couple of percent.

Numerous parameters can be varied in assessing the benefits of deltas in this context, and we have evaluated several. The results do not appear to be sensitive to the size of shingles or the delta-encoding algorithm, within reason. The extent of the match of the number of features is a good predictor of the delta size. Perhaps most importantly, when multiple files match the same number of features, there is minimal difference between the *best* delta—the smallest delta obtained across all the files—and the *average* delta. The latter two results suggest that while it is beneficial to determine the file(s) with the maximal number of matching features, only one delta need be computed. This is crucial because finding matching features, given a precomputed database of the features of other files and the dynamically computed feature set of the file being delta-encoded, is far more efficient than computing an actual delta.

2 Framework

This section describes our approach to the problem of delta-encoding with resemblance detection in greater detail. We discuss the types of data we considered and the way in which we evaluate the potential benefits of DERD.

2.1 Types of Data

In the past, delta-encoding has been used for many types of data in numerous environments. Our interest has fo-

cused on data that are located “together,” meaning that they belong to a single user, or they reside on a single server. Earlier work has demonstrated the potential benefits of deltas when the same object is modified over time, whereas we consider different objects that exist at the same time. Thus far, we have analyzed web data (primarily HTML), email, and a file system.

In a Research Report [10] coauthored with Kiem-Phong Vo of AT&T Labs, we previously argued that one could use Broder’s technique for efficiently selecting features of objects to determine dynamically a suitable candidate to serve as the base for HTTP delta-encoding. This would be an extension to the proposed standard described in a recent RFC [15]. The report described a possible protocol but gave no statistics to support the utility of the idea in practice. In the case of individual web clients, objects must be large enough to justify the added overheads of transmitting their features, comparing the features on a client, possibly computing a new delta-encoding on the fly in response to the client’s request, and reconstructing the page on the client. Beyond that proposal, similarity among different web pages could be used for efficient distribution of new pages to caches in a content distribution network (CDN), or other replicas; in this case, by transmitting many pages at once, overheads could be minimized. We have estimated the best-case benefits for a web-based DERD system, by downloading numerous pages from several sites at a single point in time, and then comparing each page against the others. In practice, not all the other pages would be cached by an individual client, though they might be cached by a CDN if they are not completely dynamic.

In parallel with assessing the overlap of content on real web sites, we identified the overlap of content in email and other local file system content as an appropriate application domain. At any instant, all the files are available, so in theory any file could be represented as a delta from one or more other files. As new files are created, they could be encoded against all earlier stored files, especially a previous version of the same file should it exist. If a “live” file system uses this approach, it must use techniques such as copy-on-write and reference counting to ensure that the base version against which a delta was computed is not modified or deleted until the delta itself is no longer needed. The same approach could be used to efficiently back up a file system: rather than delta-encoding updates in an incremental backup, the entire file system would be compressed by identifying where similarity exists.

None of these techniques would be useful without significant reduction in file sizes, so the primary focus of this study is to evaluate those reductions. Like the earlier study of deltas in HTTP [16], we consider regular compression as a basis for comparison, since compress-

ing each object to remove internal redundancy is trivial. We analyzed several datasets: the contents of `/usr` on a Redhat Linux 7.1 PC, totaling nearly 2 Gbytes of data; the contents of a user’s MH mail repository, with each message stored in a separate file (possibly including one or more MIME attachments) totaling 566 Mbytes of data; and the contents of several users’ Lotus Notes mail, with message bodies and attachments separated into distinct files. Section 3 describes the datasets in detail.

2.2 Evaluation Metrics and Practical Considerations

As noted above, size reduction is the crucial determining factor for the success of our proposal. This reduction must be considered not only relative to the original content, but relative to the size of the content using traditional compression tools such as *gzip*. Considering that reconstructing the original requires the reference file to be available, one might favor a compressed version over a delta-encoded version if the former is marginally larger.

Furthermore, the effect of the reduction is dependent on the environment:

- If an individual file is encoded, either as a delta or simple compression, and then stored on disk or some other block-based medium, the gain is not exactly the number of bytes by which the file is reduced. Instead, it is a function of the number of blocks taken up by the file before and after encoding. For instance, if every file is rounded to the nearest 4-Kbyte boundary, then shrinking a file from 4097 bytes to 4095 bytes actually saves 1 block, i.e. 4096 bytes. More typically, a file might be encoded but still use the same number of blocks on disk.
- Similarly, reducing traffic over a network has low marginal benefits if the same number of packets is used; however, if the number of round-trips in communication can be decreased, the improvement in response time is more significant.
- If many files are encoded together, such as a full backup or web server replication, then the benefits are more directly related to the actual per-file gains, since rounding effects are amortized over the entire dataset.

There are other evaluation metrics of interest, including:

Computation There are overheads due to computing the features for each file, comparing the features of the candidate and stored files, and encoding a delta once a base version is selected. Since there has been extensive research in making both delta-encoding [1] and resemblance detection [5, 6] ef-

ficient even in enormous datasets such as Internet search engines, and because our prototype is geared toward assessing space reduction benefits rather than speed, we do not report timings in this paper. However, we discuss performance issues in general terms in Section 6.

Space overheads A system that is selecting a base version given a set of features must be able to compare those features to a large set of existing files. The overhead per file may be from 50-800 bytes depending on how much information is stored, which in turn affects the quality of the comparison [6].

Execution parameters There are a number of run-time parameters that can affect the performance and/or effectiveness of the system. We consider the following:

Size and number of features Shingling a file creates an enormous number of fingerprints, or features, representing sequences of data. Broder's technique selects a "small" number of them, where "small" is parameterizable [5]. We evaluated the sensitivity of the results to this parameter. We also can require a minimal fraction of features to match before computing a delta, to see if the poorer matches still demonstrate benefits. Finally, the number of bytes used to create a single feature can vary.

Best matches If multiple files match the same number of features, an exhaustive computation could determine which base file produces the smallest delta. In fact, a file matching fewer features could produce a smaller delta than one matching more features. However, in practice, one would want to consider as few base versions as possible. While it was not possible to perform an exhaustive search within large datasets, we sampled several files with an equal number of matching features to determine whether there is a significant variance among candidate base files.

There is also an interaction between the number of features and the quality of the match. If more features are compared, then different base files can be distinguished more finely, possibly resulting in a smaller delta.

Lastly, some files may produce particularly large savings relative to an entire dataset, while others may contribute relatively little. Assuming files are sorted by the savings from encoding them, we analyze how many files need be delta-encoded to produce a given fraction of the total benefit.

Unzip-Rezip A small change to a file can result in significant differences in a com-

pressed version of the file. For example, we made a copy of the Redhat 7.1 `/usr/share/dict/words` (409,276 bytes, 45,424 one-word lines) and changed line six from `abandon` to `xyzzzy`. We call the copy `words1`. Both `words` and `words1` generated gzipped files of about 131 Kbytes, with a difference of just four bytes in size. Encoding the differences between the uncompressed `words1` and `words`, using `vcdiff`, represented the differences in just 79 bytes. In stark contrast, delta-encoding `words1.gz` against `words.gz` generated about 93 Kbytes.

Therefore, delta-encoding two compressed files by encoding their uncompressed versions and compressing the result (if needed) has the potential for significant gains. Since zip can store an arbitrarily large number of files and directories as a single compressed file, comparing its contents individually and zip-ing the results into a single zip file can have similar benefits. One might assume that `tar` need not be handled specially, since it concatenates its input without compression. We find below that this hypothesis is incorrect for the three delta-encoding programs we tried. For all these datatypes, however, the overall effects depend on the mix of data: in practice, the number and size of compressed files that can benefit from this approach may be dwarfed by all the other data.

Delta-encoding algorithm and parameters

There are a few possible delta-encoding programs. We did not find significant differences in output sizes among the available programs; therefore, following the approach of delta-encoding in HTTP [16], we report numbers using Korn and Vo's `vcdiff` [13].

Delta-encoding versus compression We vary a parameter that specifies how much smaller a delta must be than simply compressing a file before the delta is used. If no delta is small enough, of the files used as potential base versions, the compressed version is used instead. We use `vcdiff` for compression (delta-encoding a file against `/dev/null`), due to historical reasons. Its data reduction is comparable to `gzip`, though typically slightly worse.

Identical files When an identical file appears multiple times in a dataset, it can be trivially encoded against another instance through the use of hash functions such as MD5. Past stud-

ies have investigated the prevalence of mirrors on the web [4] and techniques for suppressing duplicate payloads [12]. We chose to suppress duplicates from consideration in our analysis, since they are trivially handled through other means, except when a file contained in a zip archive is duplicated (since two zip files may have many identical files and some changed content, and our unzip-rezip procedure would match up the identical files).

3 Datasets

We separate our analyses into two types of data: web pages and files in a file system. We lump email into the latter category, since in general we expect the benefits to be greater for static encoding (space reduction) than network transmission. Note that not all the datasets we analyzed are discussed further in this paper, but we include them in the tables to give a sense of the variability of the results.

3.1 Web Data

Ideally, to analyze the benefits of DERD for the web, one would study a live implementation over an extended time, and/or use full content traces to simulate an implementation. The latter approach was used effectively to study delta-encoding based on identical URLs [16], but such traces are difficult to obtain.

Instead, we used the *w3get* program to download a small set of root web pages, and recursively the pages linked from them, up to two levels. We specifically excluded file suffixes that suggested image data, such as JPG and GIF, focusing instead on the base pages. This is partly because delta-encoding has already been demonstrated to be ineffective across two different image files, even having the same name [16], and partly because images change more slowly than HTML [9] and are more likely to be cached in the first place.

While periodic downloads of specific web pages have been used in the past to evaluate delta-encoding [13], cross-page comparisons require a single snapshot of a large number of pages. We believe these pages, and the results obtained from them, demonstrate a high degree of overlap in content between pages on the same site; this has been observed in other research due to the high use of “templates” for creating dynamic pages [3, 23].

Table 1 lists the sites accessed, all between 24-26 July 2002, with the number of pages and total size. Note that in the case of Yahoo!, the download was aborted after about 27 Mbytes were downloaded, as that offered sufficient data to perform an analysis, and it was unclear how much additional data would be retrieved if left unchecked.

3.2 File Data

We used two types of file data, which are summarized in Table 2. First, we scanned the entire */usr* directory in a nearly unmodified Redhat Linux 7.1 distribution, totaling just under 2 Gbytes of data in over 100K files. Second, we examined email from several users and in several formats. Much of our analysis used over 500 Mbytes of one user’s UNIX-based email, which is stored individually in separate files by the MH mail system. The remaining data came from Lotus Notes, which stores message bodies and attachments as separate objects in a flat-file document database. We studied the attachments of five users and the message bodies of two.

4 Experiments

As described in Section 2.2, we varied a number of parameters in the delta-encoding and resemblance detection process. Our general goals were to determine how much more data could be eliminated by using deltas rather than just compression, and how sensitive that result would be to this set of parameters. In particular, we wanted to estimate the minimal work a system might do to get a reasonable benefit (i.e., the point of diminishing returns).

In general, we fixed the parameters to a common set. We then varied each parameter to evaluate its effect. Table 3 lists these parameters, with a brief description of each one, the default value in **boldface**, and other tested parameters. The parameters are clustered into two sets: the first controls the pass over the data to compute the features, and the second controls the comparison of those features and computation of the deltas.

In some cases, due to space constraints, we do not present additional details about variations in parameters that did not significantly affect results; these are denoted by *italic text*. Additional descriptions of many of the parameters were given above in Section 2.2. Note that *min_features_ratio* is special, in that it is possible to compute the savings for each number of matching features and then compute a cumulative benefit for each number of matches in a later stage, as demonstrated in Section 5.1.

4.1 Implementation Details

Most of the work to encode differences based on similarity is performed by a pair of Perl scripts. One of these recursively descends over a set of directories and invokes a Java program to compute the features. Each computation is a separate invocation of Java, though that could be optimized. Once a file’s features have been computed, they are cached in a separate file.

The other script takes the precomputed set of file-names and features, and for each file determines which

Name	Files From...	Files	Size (Mbytes)	Delta%	Comp%
Yahoo	yahoo.com	3,755	27.55	8	34
IBM	ibm.com	177	3.21	19	36
Masters	masters.com	192	3.19	9	35
CNN	cnn.com	73	2.53	15	29
Wimbledon	wimbledon.com	190	2.40	10	35

Table 1: Web datasets evaluated. Delta and compression percentages refer to the size of the encoded dataset relative to the original.

Name	Files From...	Files		Size (Mbytes)	Delta%	Comp%
		Included	Excluded			
/usr	/usr	102,932	1,250	1,964.16	36	45
MH	one user's MH directory	87,005		565.69	34	54
User1_Bod	User 1's Notes mail bodies	3,097		5.97	29	60
User1_Att	User 1's Notes mail attach.	189		81.29	71	75
User2_Bod	User 2's Notes mail bodies	445		1.18	42	56
User2_Att	User 2's Notes mail attach.	1,078		417.35	32	37
User3_Att	User 3's Notes mail attach.	140		36.18	52	61
User4_Att	User 4's Notes mail attach.	1,982		991.45	53	66

Table 2: File datasets evaluated. Excluded files are explained in the text. Delta and compression percentages refer to the size of the encoded dataset relative to the original.

other files have the maximum number of matching features. Currently this is done by identifying which features a file has, and incrementing counters for all other files with a given feature in common, using the value of the feature as a hash key. This records the most features in common at any point, F . After all features are processed, any files that have at least one feature in common are sorted by the number of matching features. Typically, only the files that match exactly F features are considered as base versions, up to the `max_comparisons` parameter, but if the best matches fail to produce a small enough delta, poorer matches are considered until the maximum is reached. There are methods to optimize this comparison by precomputing the overlap of files, as well as through estimation [22], which we intend to integrate at a later date.

Delta-encoding is performed by one of a set of programs, all written in C. Once a pair of files has been so encoded, the size of the output is cached. Occasionally, the delta-encoding program might generate a delta that is larger than the compressed file, or even larger than the original file. In those cases, the minimum of the other values is used.

For a given dataset, the results are reported by listing how many files have a maximum features match for a given number of features, with statistics aggregated over those files: the original size, the size of the delta-encoded output, and the size of the output using vc-

diff compression (delta-encoding against `/dev/null`, comparable to *gzip*). Table 4 is an example of this output. The rows at the top show dissimilar files, where deltas made no difference, while the rows at the bottom had the greatest similarity and the smallest deltas. The `BestDelta` and `AvgDelta` columns show that, in general, there was at most a 1% difference in size (relative to the original file) between the best of up to ten matching files and the average of all ten. This characteristic was common to all the datasets. Correspondingly, in all the figures, the curves for the savings for delta-encoding depict the average cases.

There are two apparent anomalies in Table 4 worth noting. First, there is a substantial jump in size at the complete 30/30 features match, despite a consistent number of files, showing a much higher average file size. This is skewed by a large number of nearly identical files, resulting from form letters attaching manuscripts for review; if each manuscript was sent to three persons and the features in the large common data were all selected by the minimization process, they all match in every feature. (This is a desirable behavior, but may not be typical of all datasets.) Second, the files with 0-2 out of 30 features matching have a dramatically worse compression ratio than the other data. We believe these are attributable to types of data that neither match other files to a great extent nor exhibit particularly good compressibility from internally repeated text

Processing Stage	Parameter	Description	Values
Preprocessing	shingle_size	Number of bytes in a fingerprinted shingle	20 , 30
	num_features	Number of features compared	30 , 100
	min_size	Minimum size of an individual file to include in statistics	128 , 512 bytes
	unzip	Should zip files be unzipped before comparison	yes, no
	gunzip	Should gz files be unzipped before comparison	yes, no
Encoding	static_files	Whether encoding A against B precludes encoding B against A	web=no, files=yes
	program	Program to perform delta-encoding	vcdiff
	exhaustive_search	Whether to compare against all files, or just best matches	no, yes
	max_comparisons	Maximum number of files to compare against, with equal maximal matching features	10 , 1, 5
	min_features_ratio	What fraction of features must match to compute a delta?	0-1 (cumulative distribution)
	improvement_threshold	What is the maximum size of a delta, relative to simple compression, for it to be used?	25%, 50%, 75%, 100%

Table 3: Parameters evaluated. **Boldface** represents defaults, and *italics* represent evaluated cases not reported here.

Matches	Files	Size (Mbytes)	BestDelta (%)	AvgDelta (%)	Compressed (%)
0	230	4.37	65	65	65
1	2634	95.09	64	65	65
2	3308	63.87	58	58	60
3	3927	30.86	39	40	45
4	4284	32.53	31	32	39
5	4710	22.86	35	36	46
			...		
27	294	2.85	4	4	46
28	227	3.09	2	2	44
29	174	9.39	0	0	43
30	224	91.38	0	0	48
All	87005	565.69	34	34	54

Table 4: Delta-encoding and compression results for the MH directory. Percentages are relative to original size, e.g. 34% means deltas save about two-thirds of the original size. Boldfaced numbers are explained in the text. This table corresponds to the graphical results in Figure 1.

strings. MIME-encoded compressed data would have this attribute, when the same compressed file does not appear in multiple messages.

To analyze the benefits of unzipping files, encoding them, and zipping the results, we take two approaches. Zip files can contain entire directory hierarchies, while gzip files compress just one file. Therefore, for zip

files, we create a special *ZIPDIR* directory, into which the contents are *unzipped* before features are calculated. We assume there are no additional benefits to compression, since zip has already taken care of that. For deltas, we delta-encode each file in this directory, storing the results in a second temporary directory, and then zip the results. For *gzip* files, we *gunzip* the files, compute

the features, and discard the uncompressed output. Each time we delta-encode a gzipped file, either as the reference or the version, we uncompress it on the fly (the most recent uncompressed version file is then cached and reused for each encoding). Section 5.4 discusses the added benefits of these two approaches.

In some cases, the features for all the files in a single dataset, with other run-time state, resulted in a virtual memory image that exceeded the 512 Mbytes of physical memory on the machine performing the comparisons—this is an artifact of our Perl-based prototype, and not inherent to the methodology, as evidenced by the scale of the search engines that use resemblance detection to suppress duplicates [6]. For the `usr` and `MH` datasets, we preprocessed the data to separate them into manageable subdirectories, then merged the results. This would result in files in different partitions not being compared: for example, a file in `Mail/conferences` would not be compared against a file in `Mail/projects`. In general, spatial locality would suggest that the best matches for a file in `Mail/conferences` would be found in `Mail/conferences`. (We subsequently validated this theory by rerunning the script on all `MH` directories at once, using a more capable machine, with no significant difference in the overall benefits.) Also, since partitions were based on subdirectories of a single root such as `/usr`, it also would result in some partitions having too few files to perform meaningful comparisons; we skipped any subdirectories with fewer than 100 files, resulting in a small fraction of files being omitted (listed in Table 2).

5 Results

Here we present our analyses. We start with overall benefits for different types of data, then describe how varying certain parameters impacts the results.

5.1 Overall Benefits

Our overall goal is to reduce file sizes and to evaluate how sensitive this reduction is to different data types, the amount of effort expended, and other considerations. Table 4 gives a sense of these results, in tabular form, for a dataset that is particularly conducive to this approach; Figure 1 shows the same data graphically. Figure 1(a) plots compressed sizes and delta-encoded sizes, as well as the original total file sizes, against the number of matching features. For each possible number of matching features from 0–30, we plot the total data of files having that number of matching features as their maximum match. As we expected, the more features match, the smaller the delta size. The cumulative effect is shown in Figure 1(b). In this graph (as well as several subsequent ones with the same label on the X-axis), a point (X,Y) shows that the total data size ob-

tained using a particular technique such as compression or delta-encoding is Y if all files with at least X maximal matching features are encoded. For instance, the Y -value of the point on the *Compressed* curve with X -value 15 is the percent of the total data size obtained if all files matching at least one other file in at least 15 features are compressed. Figure 1(b) shows that the most benefit is derived from including all files, even with zero matches, although in those cases these benefits come from compression rather than deltas—recall that the size of a delta is never larger than delta-encoding it against the empty file, i.e., compressing it.

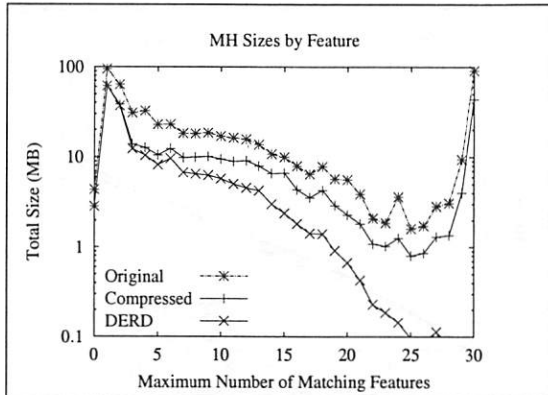
Figure 2(a) shows the cumulative benefits of deltas and compression for two of the static datasets: `usr`, and the `MH` data. Figure 2(b) does the same for two of the web datasets, `IBM` and `Yahoo`. Both graphs are limited to two datasets in order to avoid cluttering them with many overlapping lines, but the bottom-line savings for the other datasets were reported in Table 2 and Table 1, respectively. In each, the different datasets show different benefits, due to the amount of data being compared and the nature of the contents. Specifically, the graphs have very different shapes because many more files in the web datasets have high degrees of overlap.

5.2 Contributions of Large Files

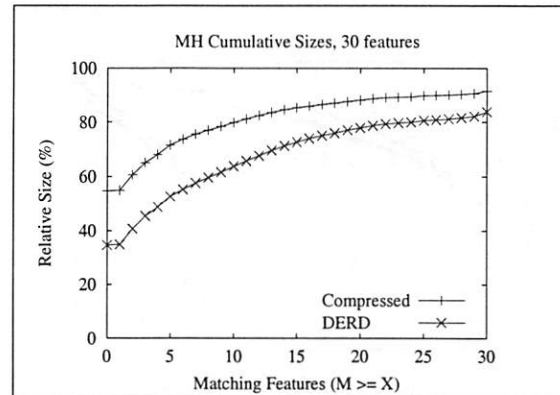
The graphs presented thus far have emphasized the effect of statistics such as the number of features that match. Another consideration is the skew in the savings: *do a small number of files contribute most of the benefits of delta-encoding?* In the case of the `MH` dataset, such a skew was suggested by the statistics in Table 4, which showed 91 of the 566 Mbytes matching in all 30 features and delta-encoding to virtually nothing.

We visualize an answer to this question by considering every file in a particular dataset, sorting by the most bytes saved for any delta obtained for it, and plotting the cumulative distribution of the savings as a function of the original files. Figure 3(a) plots the cumulative savings of the `MH` dataset (as a fraction of the original data) against the fraction of *files* used to produce those savings or the fraction of *bytes* in those files. In each case the savings for `DERD` and strict compression are shown as separate curves. Finally, points are plotted on a log-log scale to emphasize the differences at small values, and note that the `Comp by byte%` curve starts at just over 2% on the X -axis.

The results for this dataset clearly show significant skew. For example, for deltas, 1% of the files account for 38% of the total 65% saved; encoding 25% of the bytes will save 22% of the data. Compression also shows some skew, since some files are extremely compressible. If one compressed the best files containing 25% of the bytes, one would save 17% of the data. This degree of

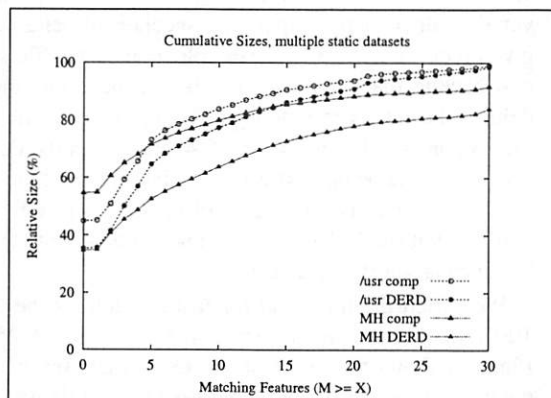


(a) Total data sizes for the original dataset, using compression, and using DERD, for individual numbers of matching features. Most of the data match very few features in any other file, or match all the features. The y-axis is on a log scale.

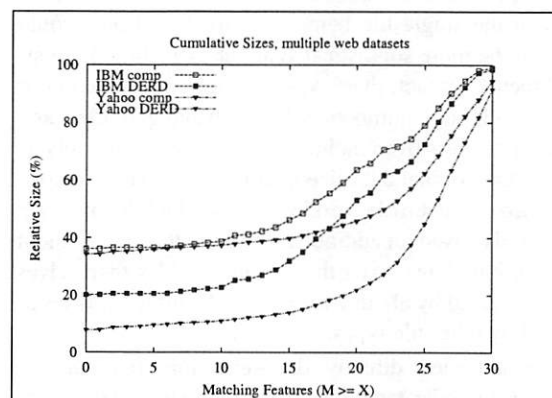


(b) Cumulative benefits. The y-axis shows the relative size, in percent, of compressing or delta-encoding each file. A point on the x-axis shows the benefit from performing this on all files that match at least that many features.

Figure 1: Effect of matching features, for the MH data. These figures graphically depict the data in Table 4.



(a) Static datasets.



(b) Web datasets.

Figure 2: Effect of matching features, cumulative, for several datasets.

skew suggests that heuristics for intelligently selecting a subset of potential delta-encoded pairs, or compressed files, could be quite beneficial.

5.3 Effects of File Blocking

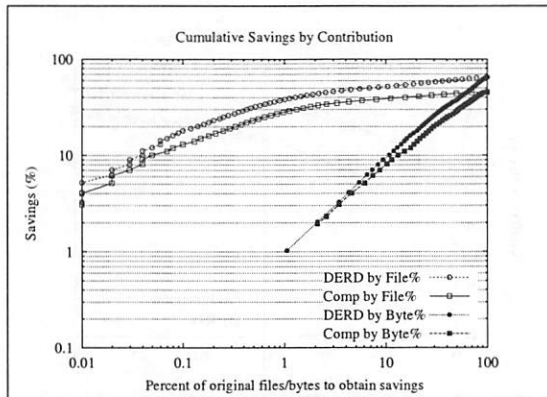
Section 2.2 referred to an impact on size reduction from rounding to fixed block sizes. In some workloads, such as file backups, this is a non-issue, but in others it can have a moderate impact for small blocks and a substantial impact for large ones.

Figure 3(b) shows how varying the blocksize affects overall savings for the MH dataset. Like Figure 3(a), it plots the cumulative savings sorted by contribution, but it accounts for block rounding effects. A 1-Kbyte min-

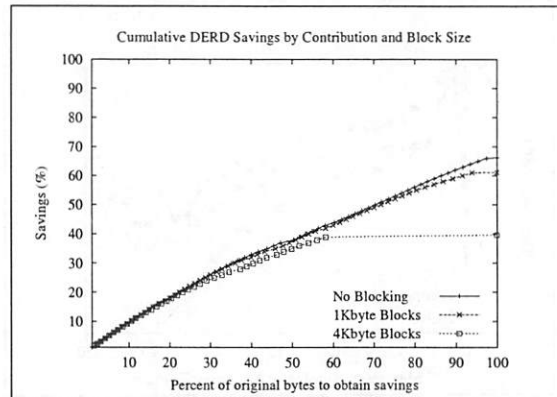
imum blocksize, typical for many UNIX systems with fragmented file blocks, reduces the total possible benefit of delta-encoding from around 66% (assuming no rounding) to 61%, but a 4-Kbyte blocksize brings the benefit down to 40% since so many messages are smaller than 4 Kbytes.

5.4 Handling Compressed and Tarred Files

Section 2.2 provided a justification for comparing the uncompressed versions of zip and gzip files, as well as a hypothesis that tar files would not need special treatment. For some workloads this is irrelevant, since for example the MH repository stored all messages with full



(a) Relative savings as a function of cumulative files, by count or by bytes. Plotted on a log-log scale.



(b) Relative savings assuming no file blocking, or rounding to 1-Kbyte or 4-Kbyte units.

Figure 3: Cumulative savings from MH files, sorted in order of contribution to total savings.

bodies, uncompressed. An attachment might contain MIME-encoded compressed files, but these would be part of the single file being examined, and one would have to be more sophisticated about extracting these attachments. In fact, there was no single workload in our study with large numbers of both zip and gzip files, and overall benefits from including this feature were only 1-2% of the original data size in any dataset. For example, the *User4.Attach* workload, which had the most zip files, only saved an additional 2% over the case without special handling. Even though the zip files themselves were reduced by about a third, overall storage was dominated by other file types.

We expected directly delta-encoding one tar file against a similar tar file to generate a small delta if individual files had much overlap, but this was not the case in some limited experiments. *Vcdiff* generated a delta about the size of the original gzipped tar file, and two other delta programs used within IBM performed similarly. We tried a sample test, using two email tar file attachments unpacked into two directories, and then using DERD to encode all files in the two directories. We selected the delta-encoded and compressed sizes of the individual files in the smaller of the tar files, and found delta-encoding saved 85% of the bytes, compared to 71% for simple compression of individual files and 79% when the entire tar file was compressed as a whole. Depending on how this extends to an entire workload, just as with zip and gzip, these savings may not justify the added effort.

5.5 Deltas versus Compression

By default, our experiments assumed that if a delta is at all smaller than just using compression, the delta is

used. There are reasons why this might not be desirable, such as a web server using a cached compressed version rather than computing a specialized delta for a given request. As another example, consider a file system backup that would require both a base file and a delta to be retrieved before producing a saved file: if the compressed version were 25% larger than the delta, it would consume that extra storage, but restoring the file would involve retrieving 125% of the delta's size rather than the delta and a base version that would undoubtedly be much larger than that 25%.

We varied the threshold for using a delta to be 25-100% of the compressed size, in increments of 25%. Figure 4 shows the result of this experiment on the MH dataset. There is a dramatic increase in the relative size of the delta-encoded data at higher numbers of matching features, because in some cases, there is no longer a usable match at a given level. The most interesting metric is the overall savings if all files are included, since that no longer suffers from this shift; the relative size increases from about 35% to about 45% as the threshold is reduced.

5.6 Shingle Size

Unlike some of the other parameters, the choice of shingle size—within reason—seems to have minimal effect on overall performance. As an example, Figure 5 shows how the size reduction varies when using shingle sizes of 20 versus 30 bytes. If all files are encoded, even for minimal matches, the total size reduction is about the same. If a higher value of *min_features_ratio* is used, the 20-byte shingles produce smaller deltas for the same threshold within a reasonable range (10-15 of 30 features matching).

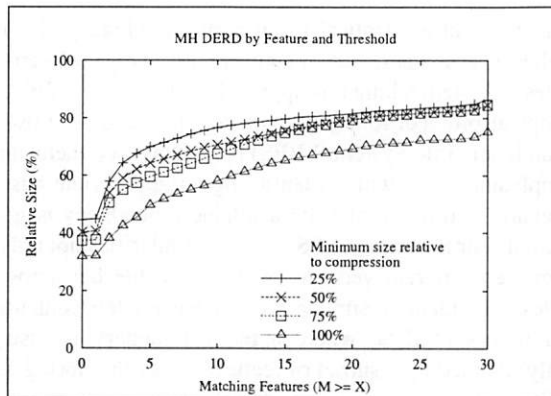


Figure 4: Effect of limiting the use of deltas to a fraction of the compressed file, for the MH dataset.

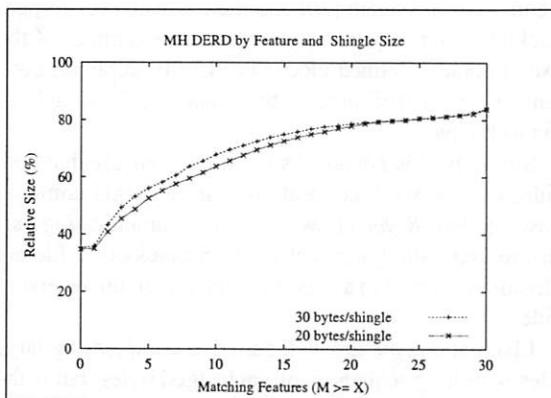


Figure 5: Effect of varying the shingle size between 20 and 30 bytes, for the MH dataset.

5.7 Number of Features

The number of features used for comparisons represents a tradeoff between accuracy of resemblance detection and computation and storage overheads. In the extreme case, one could use Manber's approach of computing and comparing every feature, and have an excellent estimate of the overlap between any two files. The other extreme is to use no resemblance detection at all or have just a handful of features. Since we have found a fair amount of discrimination using our default of 30 features, we have not considered fewer features than that, but we did compute the savings for the MH dataset from using 100 features instead of 30. The results were virtually indistinguishable in the two cases—leading to the conclusion that 30 features are preferable, due to the lower costs of storing and comparing a given number of features.

Broder has described a way to store the features even more compactly, such as 48 bytes per file, by treating the features as aggregates of multiple features computed in

the “traditional” method [6]. For one such meta-feature to match, all of some subset of the regular features must match exactly, suggesting a higher degree of overlap than we felt would be appropriate for DERD.

6 Resource Usage

A system using our techniques to efficiently delta encode files and web documents could compute features for objects when it first becomes aware of them. The cost for determining features is not that high, and it could be amortized over time. The system could also be tuned to perform delta-encoding when space is the critical resource and to store things in a conventional manner when CPU resources are the bottleneck.

Using 30 features of 4 bytes apiece, the space overhead per file is around 120 bytes. For large files, this is insignificant. Once the features for a file have been determined, it requires $O(n)$ operations to determine the maximum number of matching features with existing files where n is the total number of files. However, to get a reasonably good number of matching features, it is not always necessary to examine features for all of the existing files. A reasonable number of matching features can often be determined by only examining a fraction of the objects when the number of objects is large. That way, the number of comparisons needed for performing efficient delta-encoding can be bounded.

Delta-encoding itself has been made extremely efficient [1], and it should not usually be a bottleneck except in extremely high-bandwidth environments. Early work demonstrated its feasibility on wireless networks [11] and showed that processors an order of magnitude slower than current machines could support deltas over HTTP over network speeds up to about T3 speeds [16]. More recent systems like *rsync* [26] and LBFS [17], and the inclusion of the Ajtai delta-encoding work in a commercial backup system, also support the argument that DERD will not be limited by the delta-encoding bandwidth.

7 Related Work

Mogul, et al., analyzed the potential benefits of compression and delta-encoding in the context of HTTP [16]. They found that delta-encoding could dramatically reduce network traffic in cases where a client and server shared a past version of a web page, termed a “delta-eligible” response. When a delta was available, it reduced network bandwidth requirements by about an order of magnitude. However, in the traces evaluated in that study, responses were delta-eligible only a small fraction of the time: 10% in one trace and 30% in the other, but the one with 30% excluded binary data such as images. On the other hand, most resources were compressible, and they estimated that compressing those re-

sources dynamically would still offer significant savings in bandwidth and end-to-end transfer times—factors of 2-3 improvement in size were typical.

Later, Chan and Woo devised a method to increase the frequency of delta-eligible responses by comparing resources to other cached resources with similar URLs [7]. Their assumption was that resources “near” each other on a server would have pieces in common, something they then validated experimentally. They also described an algorithm for comparing a file against several other files, rather than the one-on-one comparison typically performed in this context. However, they did not explain how a server would select the particular related resources in practice, assuming that it has no specific knowledge of a client’s cache. We believe there is an implicit assumption that this approach is in fact limited to “personal proxies” with exact knowledge of the client’s cache [11, 2], in which case it has limited applicability.

Ouyang, et al., similarly clustered related web pages by URL, and tried to select the best base version for a given cluster by computing deltas from a small sample [18]. While they were not focused on a caching context, and are more similar to the general applications described herein, they did not initially use the more efficient resemblance detection methods of Manber and Broder to best select the base versions. Subsequently, they applied resemblance detection techniques to scale the technique to larger collections [19]. This work, roughly concurrent with our own, is similar in its general approach. However, the largest dataset they analyzed was just over 20,000 web pages, and they did not consider other types of data such as email. Another possibly significant distinction is that they used shingle sizes of only 4 bytes, whereas we used 20-30 bytes. (We did not obtain this paper in time to repeat our analyses with such a small shingle size.)

Spring and Weatherall [24] essentially generalized Chan and Woo’s work by applying it to all data sent over a specific communication channel, and using resemblance detection to detect duplicate sequences in a collection of data. This was done by computing fingerprints of shingles, selecting those with a predetermined number of zeroes in the low-order bits (deterministically selecting a fraction of features), and scanning before and after the matching shingle to find the longest duplicate data sequence. Like Chan and Woo’s work, this system worked only with a close coupling between clients and servers, so both sides would know what redundant data existed in the client. In addition, the communication channel approach requires a separate cache of packets exchanged in the past, which may compete with the browser cache and other applications for resources.

In some cases, the suppression of redundancy is at a very coarse level, for instance identifying when an en-

tire payload is identical to an earlier payload [12], or when a particular region of a file has not changed. Examples of system taking this approach include *rsync* [26], a popular protocol for remote file copying, and the Low-bandwidth File System (LBFS) [17]. However, there are applications for which identifying an appropriate base version is difficult and the available redundancy is ignored. For instance, LBFS exploits similarities not only between different versions of the same file but across files. To identify similar files, it hashes the contents of blocks of data, where a block boundary is (usually) defined by a subset of features—like the Spring & Weatherall approach, except that the features determine block boundaries rather than indices for the data being compared. Variable block boundaries allow a change within one block not to affect neighboring blocks. (The Venti archival system [20] and the Pastiche peer-to-peer backup system [8] are two more recent examples of the use of content-defined blocks to identify duplicate content; we use LBFS here as the “canonical” example of the technique.)

Similarly, it is not always possible to ensure that both sides of a network connection share a single common base version. *Rsync* allows the two communicating parties to ascertain dynamically which blocks of a file are already contained in a version of the file on the receiving side.

LBFS and *rsync* are well suited to compressing large files with long sequences of unchanged bytes, but if the granularity of change is finer than their block boundaries, they get no benefit. Most delta-encoding algorithms remove redundancy if it is large enough to amortize the overhead of the pointers and other meta-data that identify the redundancy. A resemblance detection procedure should therefore be suited to the delta-encoding algorithm, and the size and contents of the data. Our work demonstrates that fine-grained deltas work well in a variety of environments, but a head-to-head comparison with LBFS and *rsync* in these environments will help determine which approach is best in which context.

8 Conclusions and Future Work

Delta-encoding has been used in a number of applications, but it has been limited to two general contexts: encoding a file against an earlier version of the same file, or encoding against other files (or data blocks) where both sides of a communication channel have a consistent view of the cached data. We have generalized this approach in the web context to use features of web content to identify appropriate base versions, and quantified the potential reductions in transfer sizes of such a system. We have also extended Manber’s use of this technique on a single server [14], and quantified potential benefits in a general file system and specific to email.

For web content, we have found substantial overlap among pages on a single site. This is consistent with Chan and Woo [7], Ouyang, et al. [19], and recent work on automatic detection of common fragments within pages [23]. For the five web datasets we considered, deltas reduced the total size of the dataset to 8–19% of the original data, compared to 29–36% using compression. For files and email, there was much more variability, and the overall benefits are not as dramatic, but they are significant: two of the largest datasets reduced the overall storage needs by 10–20% beyond compression. There was significant skew in at least one dataset, with a small fraction of files accounting for a large portion of the savings. Factors such as shingle size and the number of features compared do not dramatically affect these results. Given a particular number of maximal matching features, there is not a wide variation across base files in the size of the resulting deltas.

A new file will often be created by making a small number of changes to an older file; the new file may even have the same name as the old file. In these cases, the new file can often be delta-encoded from the old file with minimal overhead. For the most part, our datasets did not consider these scenarios. For situations where this type of update is prevalent, the benefits from delta-encoding are likely to be higher.

Now that we have demonstrated the potential savings of DERD, in the abstract, we would like to implement underlying systems using this technology. The smaller deltas for web data suggest that an obvious approach is to integrate DERD into a web server and/or cache, and then use a live system over time. However, supporting resemblance-based deltas in HTTP involves extra overheads and protocol support [10] that do not affect other applications such as backups. We are also interested in methods to reduce storage and network costs in email systems, and hope to implement our approach in commonly used mail platforms. As the system scales to larger datasets, we can add heuristics for more efficient resemblance detection and feature computation. We can also evaluate additional application-specific methods, such as encoding individual elements of tar files, and compare the various delta-based approaches against other systems such as LBFS and *rsync* in greater depth.

Acknowledgments

Kiem-Phong Vo jointly developed the idea of web-based DERD, resulting in a research report [10] from which a small amount of the text in this manuscript has been taken. Andrei Broder has been extremely helpful in understanding the intricacies of resemblance detection, Randal Burns and Kiem-Phong Vo have similarly been helpful in providing and helping us to understand their delta-encoding software packages, and Laurence Marks

is a LotusScript guru extraordinaire. Ziv Bar-Yossef, Sridhar Rajagopalan, and Lakshmish Ramaswamy provided code for computing features. Several people have permitted us to analyze their data, including Lisa Amini, Frank Eskesen and Andy Walter. Ramesh Agarwal, Andrei Broder, Ron Fagin, Chris Howson, Ray Jennings, Jason LaVoie, Srini Seshan, John Tracey, and Andrew Tridgell have provided helpful comments on some of the ideas presented in this paper and/or earlier drafts of this paper. Finally, we thank the anonymous reviewers and our shepherd, Darrell Long, for their advice and feedback.

References

- [1] M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer. Compactly encoding unstructured input with differential compression. *Journal of the ACM*, 49(3):318–367, May 2002.
- [2] Gaurav Banga, Fred Dougli, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *Proceedings of 1997 USENIX Technical Conference*, pages 289–303, January 1997.
- [3] Ziv Bar-Yossef and Sridhar Rajagopalan. Template detection via data mining and its applications. In *Proceedings of the Eleventh International Conference on World Wide Web*, pages 580–591. ACM Press, 2002.
- [4] K. Bharat and A. Broder. Mirror, mirror on the web: A study of host pairs with replicated content. In *Proceedings of the 8th International World Wide Web Conference*, pages 501–512, May 1999.
- [5] Andrei Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, 1997.
- [6] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *Combinatorial Pattern Matching, 11th Annual Symposium*, pages 1–10, June 2000.
- [7] Mun Choon Chan and Thomas Y. C. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proceedings of Infocom'99*, pages 117–125, 1999.
- [8] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 285–298. USENIX, December 2002.
- [9] Fred Dougli, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the World Wide Web.

- In *Proceedings of the Symposium on Internet Technologies and Systems*, pages 147–158. USENIX, December 1997.
- [10] Fred Douglass, Arun K. Iyengar, and Kiem-Phong Vo. Dynamic suppression of similarity in the web: a case for deployable detection mechanisms. Technical Report RC22514, IBM Research, July 2002.
 - [11] Barron C. Housel and David B. Lindquist. Web-Express: A system for optimizing Web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 108–116. ACM, November 1996.
 - [12] Terence Kelly and Jeffrey Mogul. Aliasing on the World Wide Web: Prevalence and Performance Implications. In *Proceedings of the 11th International World Wide Web Conference*, May 2002.
 - [13] David G. Korn and Kiem-Phong Vo. Engineering a differencing and compression data format. In *Proceedings of the 2002 Usenix Conference*. USENIX Association, June 2002.
 - [14] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, January 1994.
 - [15] J. Mogul, B. Krishnamurthy, F. Douglass, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. *Delta encoding in HTTP*, January 2002. RFC 3229.
 - [16] Jeffrey Mogul, Fred Douglass, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of ACM SIGCOMM'97 Conference*, pages 181–194, September 1997.
 - [17] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.
 - [18] Zan Ouyang, Nasir Memon, and Torsten Suel. Using delta encoding for compressing related web pages. In *Data Compression Conference*, page 507, March 2001. Poster.
 - [19] Zan Ouyang, Nasir Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *International Conference on Web Information Systems Engineering (WISE)*, December 2002.
 - [20] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, Monterey, CA, 2002.
 - [21] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
 - [22] Sridhar Rajagopalan, 2002. Personal Communication.
 - [23] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglass. Techniques for efficient detection of fragments in web pages. Manuscript, November 2002.
 - [24] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, August 2000.
 - [25] W. Tichy. RCS: a system for version control. *Software—Practice & Experience*, 15(7):637–654, July 1985.
 - [26] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

Opportunistic Use of Content Addressable Storage for Distributed File Systems

Niraj Tolia^{†‡}, Michael Kozuch[‡], Mahadev Satyanarayanan^{†‡}, Brad Karp[‡],
Thomas Bressoud^{‡*}, and Adrian Perrig[†]

[†]Intel Research Pittsburgh, [‡]Carnegie Mellon University, and ^{*}Denison University

Abstract

Motivated by the prospect of readily available Content Addressable Storage (CAS), we introduce the concept of file *recipes*. A file's recipe is a first-class file system object listing content hashes that describe the data blocks composing the file. File recipes provide applications with instructions for reconstructing the original file from available CAS data blocks. We describe one such application of recipes, the CASPER distributed file system. A CASPER client opportunistically fetches blocks from nearby CAS providers to improve its performance when the connection to a file server traverses a low-bandwidth path. We use measurements of our prototype to evaluate its performance under varying network conditions. Our results demonstrate significant improvements in execution times of applications that use a network file system. We conclude by describing fuzzy block matching, a promising technique for using *approximately matching* blocks on CAS providers to reconstitute the *exact* desired contents of a file at a client.

1 Introduction

The exploding interest in distributed hash tables (DHTs) [6, 26, 28, 34] suggests that *Content Addressable Storage (CAS)* will be a basic facility in future computing environments. In this paper we show how CAS can be used to improve the performance of a conventional distributed file system built on the client-server model. NFS, AFS and Coda are examples of distributed file systems that are now well-entrenched in many computing environments. Our goal is to improve client performance in situations where a distant file server is accessed across a slow WAN, but one or more *CAS providers* that export a standardized CAS interface are located nearby on a LAN.

The concept of a file *recipe* is central to our approach. The recipe for a file is a synopsis that contains a list of data block identifiers; each block identifier is a cryptographic hash over the contents of the block. Once the data blocks identified in a recipe have been obtained, they can be combined as prescribed in the recipe to reconstruct the file. On a cache miss over a low-bandwidth network, a client may request a file's recipe rather than its data. Often, the client may be able to reconstruct the file from its recipe by contacting nearby CAS providers (including CAS services available on

the client) to which it has LAN access. In this usage scenario, a recipe helps transform WAN accesses into LAN (or local client) accesses. Since a recipe is a first class entity, it can be used as a substitute for the file in many situations. For example, if space is tight in a file cache, files may be replaced by the corresponding recipes, which are typically much smaller. This is preferable to evicting the file entirely from the cache because the recipe reduces the cost of the cache miss resulting from a future reference to the file. If the client is disconnected, the presence of the recipe may make even more of a difference — replacing an unserviceable cache miss by file reconstruction.

It is important to note that our approach is *opportunistic*: we are *not* dependent on CAS for the correct operation of the distributed file system. The use of recipes does not in any way compromise attributes such as naming, consistency, or write-sharing semantics. Indeed, the use of CAS is completely transparent to users and applications. When reconstructing a file, some blocks may not be available from CAS providers. In that case, those blocks must be fetched from the distant file server. Even in this situation, there is no loss of consistency or correctness.

CAS providers can be organized peer-to-peer networks such as Chord [34] and Pastry [28], but may also be impromptu systems. For example, each desktop on a LAN could be enhanced with a daemon that provides a CAS interface to its local disk. The CAS interface can be extremely simple; in our work, we use just four calls *Query*, *MultiQuery*, *Fetch*, and *MultiFetch* (explained in more detail in Section 5.3). With such an arrangement, system administrators could offer CAS access without being required to abide by any peer-to-peer protocol or provide additional storage space. In particular, CAS providers need not make any guarantees regarding content persistence or availability.

As a proof of concept, we have implemented a distributed file system called CASPER that employs file recipes. We have evaluated CASPER at different bandwidths using a variety of benchmarks. Our results indicate that substantial performance improvement is possible through the use of recipes. Because CASPER imposes no requirements regarding the availability of CAS providers, they are free to terminate their service at any time. This encourages users to offer their desktops as CAS providers with the full confidence that they can withdraw at any time. In some envi-

ronments, a system administrator may prefer to dedicate one or more machines as CAS providers. Such a dedicated CAS provider is referred to as a *jukebox*.

After summarizing related work in Section 2, we discuss the utility of recipes and propose an XML representation in Section 3. We next describe the architecture of CASPER in Section 4, and relate details of its implementation in Section 5. In Section 6, we evaluate the performance of CASPER and quantify its performance dependence on the availability of matching CAS blocks. We briefly review security considerations associated with using recipes in Section 7. We propose a technique in Section 8 for using similar blocks, not only exactly matching blocks, with CAS. We summarize our findings and conclude in Section 9.

2 Related Work

The use of content hashes to represent files in file systems has been explored previously. Single Instance Storage [2] uses content hashes for compression, to coalesce duplicate files, and Venti [24] employs a similar approach at the block level. The Low Bandwidth File System (LBFS) [19] operates on variable-length blocks, and exploits commonality between distinct files and successive versions of the same file in the context of a distributed file system. The Pastiche backup system [7] employs similar mechanisms. In all these systems, content hashes are used internally. The CASPER file system described herein uses similar techniques, but proposes a canonical representation of hash-based file descriptions (recipes), and promotes those descriptions to members of a first-class data type. CASPER introduces the concept of a portable recipe that can be used to support legacy applications. In addition, CASPER allows all nodes with storage to participate as CAS providers, whereas prior systems, such as LBFS, restrict their search for file system commonality to the individual peer client and server.

Much attention has been devoted to using overlay networks to form Distributed Hash Tables (DHTs). Recent work in this area includes Chord [34], Pastry [28], CAN [26], and Freenet [6]. We believe that these DHTs are very valuable in the CAS context, where they may act as content providers for our system.

File systems built atop these DHTs include CFS [8], PAST [11], and Ivy [20]. These systems, while completely decentralized, share the model that participants that join the overlay offer write access to their local storage to other participants, either directly or indirectly. In CAS, participants may offer to share the contents of their storage with others *without* agreeing to store others' data. Moreover, a CASPER client can function without connectivity to a widely dispersed collection of overlay members. A CASPER client always can fall back to requesting data from its file servers, and choose to exploit available DHT infrastructure when beneficial. With the exception of Ivy, peer-to-peer file systems have traditionally been read-only or single-publisher

systems. CASPER, however, can be layered over any traditional network file system, without changing the original file system's semantics.

Both Fluid Replication [15] and Pangaea [30] also attack the distributed file system performance problem on wide-area networks. Fluid Replication differs from CASPER in its dependence on dedicated machines to aid update propagation; while not implemented in CASPER, these techniques could complement our approach. While CASPER provides a conventional access control model in a client-server architecture, Pangaea uses a decentralized model and is designed to allow for ad hoc sharing and replication of data.

There is also a significant body of work in the area of delta encoding [18, 36, 37], but these methods require a fixed reference point for data comparison. CASPER, on the other hand, is opportunistic, does not need fixed file references, and works correctly in the absence of any previous version of an object.

3 Recipes

File recipes provide instructions for the construction of files from CAS data blocks. A recipe lists the addresses of CAS blocks that compose the desired file and describes the arrangement of those component blocks. For example, a file object could be divided into a sequence of 4 KB blocks. By listing the SHA-1 [22] hash of each block in order, we derive one possible recipe for the file. Other possible recipes exist: a sequence of SHA-1 hashes of 8 KB blocks or a sequence of hashes of variably sized blocks (such as might be generated using Rabin fingerprints [19, 25]). Once an object's recipe is known, the object may be reconstructed by fetching the component CAS objects (the "ingredients") named in the recipe from any available source and combining them as specified in the recipe.

Further, a recipe may include multiple recipe choices. Each choice is one possible method for describing the original file. With more than one choice available, a recipe-based application (*e.g.*, the CASPER file system) may select the most appropriate recipe choice for the situation. Suppose, for example, that CAS providers on one campus only support SHA-1 hashes of 4 KB blocks while CAS providers on a neighboring campus only support MD5 [27] hashes on 8 KB blocks. If a file recipe contains two choices, one based on SHA-1, 4 KB hashes and the other based on MD5, 8 KB hashes, the corresponding file could be reconstructed by recipe-based applications on either campus.

Each recipe maps a set of ingredients from the CAS namespace to a higher-level namespace, such as a file-system namespace. However, recipes are themselves first-class objects and may be stored in the higher-level namespace. One benefit of their first-class status is that recipes can be cached, and the consistency of recipes may be maintained by traditional coherency mechanisms. For example, in the CASPER file system, file recipes are stored as ordinary files

```

<?xml version="1.0"?>
<recipe type="file">
  <metadata>
    <length>125637</length>
    <last_modified>11/12/2002 16:24:37</last_modified>
    <file_system type="Coda">
      <name>/coda/projects/shared/casper.pl</name>
      <fid>312567 0 678</fid>
      <version>6 7 3 4 9</version>
    </file_system>
  </metadata>

  <recipe_choice>
    <hash_list hash_type="SHA-1" block_type="fixed"
      fixed_size="4096" number="31">
      <hash>09d2af8dd22...</hash>
      <hash>e5fa44f2b31...</hash>
      .
    </hash_list>
  </recipe_choice>

  <recipe_choice>
    <hash_list hash_type="SHA-1" block_type="variable"
      number="36">
      <hash size="3576">7448d8798a4...</hash>
      <hash size="1278">a3db5c13ff9...</hash>
      .
    </hash_list>
  </recipe_choice>

  <recipe_choice>
    <hash_list hash_type="MD5" block_type="fixed"
      fixed_size="125637" number="1">
      <hash>9c6b057a2b9...</hash>
    </hash_list>
  </recipe_choice>
</recipe>

```

Figure 1. Sample File Recipe

and may be cached in the file system cache. Consistency between cached recipes and the server version of the recipe is maintained by the cache coherency protocol. Consistency between a file and its recipe is maintained lazily by maintaining version numbers for all files.

We have adopted XML (Extended Markup Language) [4] as the language for expressing recipes. An example file recipe is shown in Figure 1. Of course, the main motivation for employing XML is portability. We believe that recipes are a generally useful abstraction, and therefore, by encoding recipes in a portable format, various applications will be able to make use of the same infrastructure.

The sample recipe in Figure 1 describes a file of 125637 bytes. While generating the recipe, the application that created the recipe included additional metadata such as the last modified time, the type of file system where the file was found, and file-system-specific data. Naturally, the recipe metadata could be extended with other information such as the file ownership and access permissions. Such information, while not required for CASPER, may benefit other applications that leverage file recipes. Following the metadata XML element are three possible recipe choices

(recipe_choice elements) for recreating the data that constitute the file.

The first recipe choice is a list of SHA-1 hashes corresponding to 4 KB blocks. In our grammar, a hash_list is used to denote a series of hashes whose contents should be concatenated to form a region. In this case, the hashes compose the entire file. In fact, the 31 4 KB blocks described in the hash list compose an object slightly larger than the length given in the metadata. In such cases, after assembly, the resultant object must be truncated to the proper-length.

The second recipe choice is a hash_list comprising SHA-1 hashes of variable-length blocks, as might be generated by employing Rabin fingerprints to find block boundaries. Again, to reconstitute the file, the blocks corresponding to the hashes in the hash list need only be concatenated to form the original file.

The third recipe choice is a hash_list comprising a single MD5 hash of the entire file. This hash may be used as a final checksum to ensure (statistically) end-to-end file integrity. When assembling a file from constituent blocks, the recipe-based application may fail to find all the requested CAS components. The missing components must be retrieved through another mechanism, and once obtained, the complete file is assembled. The file-wide hash enables the CAS application to provide confidence that the file was assembled correctly.

Note that because the various choices provided in the recipe are typically orthogonal, in some cases the CAS data can be fetched from more than one CAS source. For example, suppose that after attempting to reconstruct the file from the 4 KB hash list, the recipe-based application determines that two of the 31 blocks could not be found. The client could then query CAS providers to determine if the appropriate blocks from the variable-length list that cover the missing two blocks can be found. If so, the data from those can be used to fill in the missing regions of the file.

4 The CASPER Distributed File System

CASPER is a distributed file system that employs file recipes to reduce the volume of data transmitted from a file server to its clients. CASPER clients cache files with whole-file granularity, and relies on centralized file servers to guarantee data persistence and file consistency. The novel aspect of CASPER is that clients make opportunistic use of nearby CAS providers to improve performance and scalability.

If the available client-server bandwidth is low during a file fetch operation, the client requests a recipe rather than the contents of the file. Using the hashes contained in the recipe, CASPER attempts to reconstruct the file by fetching components from nearby CAS providers. Any components not found near the client are fetched from the CASPER server, which is responsible for maintaining a master copy of every file it serves.

Small files (our implementation classifies files smaller than 4 KB as “small files”) often do not enjoy the benefits of CAS acceleration due to recipe metadata and network overhead costs. Consequently, CASPER does not employ the recipe mechanism when transferring small files. Instead, the data composing a small file is transferred directly.

CASPER caches data at a whole-file granularity to provide the file-session oriented, open-close consistency model of AFS [13] and Coda [32, 33]. Consequently, once the file is reconstructed, it is placed in the client cache. For efficiency, CASPER clients treat their own caches as CAS providers. Before requesting a component from nearby, external CAS providers, clients inspect their own caches to determine if the component is also part of a previously cached file. In this way, CASPER mimics the fetch behavior of LBFS.

Throughout this paper, we are primarily concerned with client reads. However, we intend to extend our current implementation to accommodate client write operations by adopting a similar local cache lookup mechanism on the server-side. To leverage the recipe-based mechanism, we view client writes as server reads of the modified file. When sending file modifications to the server, a client sends a recipe of the modified file rather than the file contents. The server will then peruse its own data for instances of the components, then request components that are not found locally from nearby CAS providers, and finally retrieve any remaining components from the client directly.

As recipes in CASPER are treated as hints, the consistency between files and their recipes is managed in a lazy fashion. The file system maintains a version number for each file. When a recipe is generated, the recipe includes the version of the file from which it was derived. Because CASPER clients always check the expected version of the file against the version stored in the recipe metadata, a cached stale recipe will never be used to reconstruct a file. When a recipe is determined to be stale, CASPER triggers the creation of a new recipe by the server.

5 Implementation

Our implementation of CASPER is derived from the Coda distributed file system, and we have adopted the modular, proxy-based layering approach described in the Data Staging work [12]. Figure 2 depicts the organization of our system. The *Coda Client* and *Coda File Server* modules are unmodified releases of the Coda client and server, respectively. The *Proxy* module is responsible for intercepting communication between the client and server and determining whether or not to activate CASPER’s CAS functionality. The Coda client and proxy together act as a CASPER client. Likewise, the Coda file server and *Recipe Server* together act as the CASPER server. The recipe server is the component responsible for forming responses to recipe requests.

The proxy-based design enables us to prototype new file-system features such as the CAS-based performance en-

hancements without modifying Coda. In this design, the proxy provides the client with an interface identical to the Coda server interface. The proxy monitors network conditions and determines when to use an available CAS provider. Under optimal conditions, the CASPER system behaves identically to an unmodified Coda installation without the proxy. After detecting low bandwidth network conditions, however, the proxy registers with the *recipe server* and a CAS provider. The provider in our example is a *jukebox*.

While low-bandwidth conditions persist, the proxy intercepts all file requests from the client and asks for the corresponding recipe from the recipe server. The recipe server is responsible for generating a recipe for the current version of the file and delivering it to the proxy. The proxy then attempts to retrieve the data blocks named in the file from nearby CAS providers (including the client’s own file cache). The proxy will request that the recipe server also deliver any blocks not found on the CAS provider(s). Once the file reconstruction is complete, it is passed back to the Coda client, which places the file in its file cache. No other traffic, such as writes, is currently intercepted by the proxy; instead it passes directly to the file server.

In the next three sections, we describe the design and implementation of the recipe server, proxy and jukebox in more detail.

5.1 Recipe Server

As the name indicates, the recipe server generates the recipe representation of files. However, this module is also responsible for responding to requests for missed data blocks and forwarding callbacks.

The recipe server is a specialized user process that accesses file-system information through the standard file-system interface. Our implementation maintains generated recipes as files in the CASPER file system. For user files in a directory, *zeta*, the recipe server stores the corresponding recipes in a sub-directory of *zeta* (e.g., *zeta/.shadow/*).

In Figure 2, we show the recipe server co-located with the Coda server. However, any machine that is well-connected to the server may act as the recipe server. In fact, if a user has a LAN-connected workstation, that workstation may make an excellent choice for the location of the user’s primary recipe server because it is the most likely machine to have a warm client cache.

When a recipe request arrives at the recipe server, the recipe server first determines if the recipe file corresponding to the request exists in the file system. If so, the recipe is read, either from the recipe server’s cache or from the Coda file server, and checked for consistency. That is, the versioning information in the recipe is compared to the current version of the file. If the recipe is stale or does not exist, the recipe server will generate a new one, send it to the proxy, and store it in the shadow directory for potential reuse.

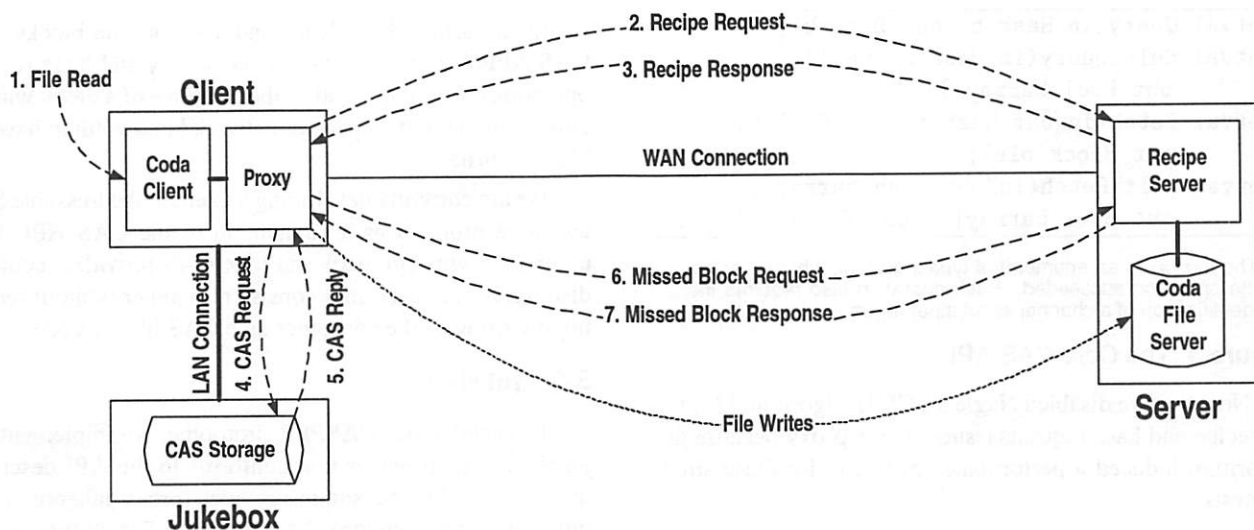


Figure 2. System Diagram

As mentioned in Section 4, the recipe server responds to requests for recipes of small files with the file's contents rather than the file's recipe. The threshold in the current implementation is 4 KB. The reason for this special handling is that the transfer time of small messages in low-bandwidth networks is often dominated by the network round-trip time.

Data blocks that are not found on any of the queried CAS providers are referred to as *missed blocks*. In our implementation, the proxy fetches missed blocks from the recipe server. Our recipe server supports requests for byte extents. That is, a missed block request is of the form `fetch(file, offset, length)`. To reduce the number of transmitted requests, missed blocks that are adjacent in the file are combined into a single extent request by the proxy, and we employ the further optimization of combining multiple fetch requests into a single multi-fetch message. In alternative implementations, missed block requests could be serviced by the file server. We chose to use the recipe server instead to reduce the file server load in installations where the recipe server is not co-located with the file server.

The recipe server also contributes to maintaining consistency by forwarding callbacks from the file server to the proxy for files that the proxy has reconstructed via recipes. That is, if the recipe server receives an invalidation callback for a file after sending a recipe for the file to a client, the callback is forwarded to the requesting client via the proxy.

Because the recipe server is a user process and not a Coda client, the recipe server does not receive the consistency callbacks directly. Instead, the recipe server communicates with the Coda client module through *coda-con* [31], Coda's standard socket-based interface, which exports callback messages. Consequently, implementation of the callback-forwarding mechanism did not require modification of the client or server. Mechanisms that gather similar information, such as the SysInternals Filemon tool [35], could potentially serve the same function should CASPER

be ported to network file systems that do not provide the necessary interface.

5.2 Proxy

The proxy is the entity that transparently adds recipe-based file reconstruction to distributed file systems without modifying the file system code. This arrangement is not necessary for the operation of CASPER but eased our implementation of the prototype. The two main tasks that it performs are fetching recipes and reconstructing files.

Because the proxy acts as the server for the client, the proxy intercepts all file read requests originating from the client. Client cache misses induce fetch requests which are caught by the proxy. The fetch operation causes the proxy to send a request for the file's recipe to the recipe server; other file system operations are forwarded directly to the Coda server. Assuming that a recipe fetch request is successful, the proxy compares the version in the received recipe to the expected version of the file. If the recipe version is more up-to-date than the proxy's expected version number, the proxy contacts the Coda server to confirm the file's version number. If the recipe version is older, there might be a problem with the recipe server (*e.g.*, the recipe server may have become disconnected from the Coda server), and therefore, the request is redirected to the file server. The request is also redirected to the file server if any unexpected recipe server errors occur.

If the version numbers match, the proxy selects one of the available recipe choices in the recipe file, and the proxy proceeds to reconstruct the file. After interpreting the XML data it received, the proxy queries both the client's local cache as well as any nearby jukeboxes for the hashes present in the recipe. Matching blocks, if any, are fetched from the jukeboxes, and the remaining blocks are fetched from the recipe server. Finally, the proxy assembles all constituent pieces and returns the resulting file to the client.

```

RetVal Query(in Hash h, out Bool b);
RetVal MultiQuery(in Hash harray[],
    out Bool barray[]);
RetVal Fetch(in/out Hash h, out Bool b,
    out Block blk);
RetVal MultiFetch(in/out Hash harray[],
    out Bool barray[], out Block blks[]);

```

The Retval is an enumerated type indicating whether or not the operation succeeded. Each operation also requires the identification of a channel as an input argument (not shown).

Figure 3. The Core CAS API

Note that we disabled Nagle's TCP/IP algorithm [21] for all recipe and hash requests issued by the proxy because the algorithm induced a performance overhead for these small requests.

5.3 Content Addressable Storage API

The CAS API defines the interface between *CAS consumers* and *CAS providers*. CAS consumers request data blocks by specifying a hash of the blocks' contents, and providers are storage devices that can respond to those requests. This API is what the proxy, a CAS consumer, uses to communicate with jukeboxes, which are CAS providers.

At the interface level, CAS consumers address providers through handles known as *channels*. This abstraction provides a common interface to all providers whether they are peer-to-peer networks, jukeboxes, or other devices. The CAS API provides a mechanism for obtaining the *characteristics* of a channel. While we do not currently use this mechanism, we envision that consumers will use it to determine such channel characteristics as the type of hashes supported, the channel's bandwidth, and some measure of the channel's topological proximity to the consumer.

The enumeration of available channels is the responsibility of a CAS API module running on the client system. The general topic of service discovery over a network is a rich area and will not be discussed here. In the context of the CAS API, the role of service discovery is to enumerate the set of available local and remote services and relay their associated characteristics to the consumer. The underlying mechanisms employed may be varied and may range from static specification and local operating system mechanisms to network-based protocols such as LDAP [14] or Universal Plug 'n Play [38].

Figure 3 summarizes the operations of the CAS API. The core operation of the CAS API is the retrieval of a CAS data block given the block's hash through the *Fetch* call. The consumer specifies the hash of the object to be retrieved (the hash data structure includes the hash type, block size, and hash value) and a channel. *Fetch* returns a boolean indicating if the block was found on the specified channel and, if so, the block itself. A consumer may instead make a single request for multiple blocks through the *MultiFetch* function, specifying an array of hashes and a channel. This function

returns an array of Booleans and a set of data blocks. The CAS API also provides associated *Query* and *MultiQuery* operations for inquiring after the presence of a block without allocating the buffer space or network bandwidth to have the block returned.

We are currently developing a Content Addressable Storage wire protocol as a companion to the CAS API. With a common wire protocol, multiple CAS providers could be discovered and used in a consistent manner without requiring rewriting of the consumer-side CAS library code.

5.4 Jukebox

To evaluate our CASPER prototype, we implemented a jukebox CAS provider that conforms to the API described in Figure 3. The consumer-provider (proxy-jukebox) communication that supports the *Query* and *Fetch* functions is currently implemented by using a lightweight RPC protocol.

The jukebox, a Linux-based workstation, uses the native *ext3fs* file system to store the CAS data blocks. Currently, the system administrator determines a set of files to be exported as CAS objects. The jukebox makes use of recipes to track the location of data blocks within the exported files. The jukebox creates an in-memory index of the data at startup. The index, keyed by the hash, provides an efficient lookup mechanism.

6 Evaluation

We measured the performance benefit of recipe-based file reconstruction using three different benchmarks. We evaluated each benchmark under several combinations of network bandwidth, network latency and jukebox hit-ratio. We describe the experimental methodology, discuss the three benchmarks, and present their results.

6.1 Experimental Methodology

Our experimental infrastructure consisted of several contemporary, single-processor machines. The jukebox and client were workstations with 2.0 GHz Pentium® 4 processors. While the client had 512 MB of SDRAM, the jukebox had 1 GB. The file server contained a 1.2 GHz Pentium® III Xeon™ processor and 1 GB of SDRAM.

The jukebox and client ran the Red Hat 7.3 Linux distribution with the 2.4.18-3 kernel, and the file server ran the Red Hat 7.2 Linux distribution with the 2.4.18 kernel. All machines ran version 5.3.19 of Coda with a large enough disk cache on the client to prevent eviction during the experiments. The recipe server process was co-located with the file server. To discount the effect of cold I/O buffer caches, we ran one trial of each experiment before taking any measurements. However, we ensured that the Coda client cache was cold for all experiments.

To simulate different bandwidths and latencies, we used the NIST Net [23] network emulator, version 2.0.12. The

client was connected to the jukebox via 100 Mb/s Ethernet, but the client's connection to the server was controlled via Nist Net. All benchmarks ran at three different client-server bandwidths: 10 Mb/s, 1 Mb/s and 100 Kb/s. We do not report results for a 100 Mb/s client-server connection, because CASPER clients should fetch data from the server directly when the client-server bandwidth is equal to, or better than, the client-jukebox bandwidth. Extra latencies of 10 ms and 100 ms were introduced for the 1 Mb/s and 100 Kb/s cases, respectively. No extra latency was introduced for the 10 Mb/s case.

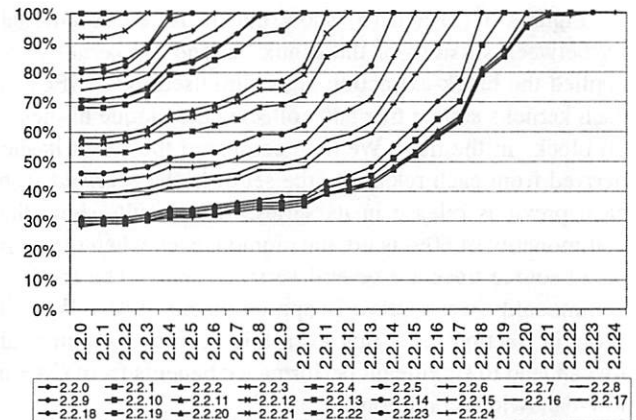
The effectiveness of recipe-based reconstruction in CASPER is highly dependent on the fraction of data blocks the client can locate on nearby CAS providers. We used sensitivity analysis to explore the effect of the CAS provider hit rate on CASPER performance. For each of the experiment and bandwidth cases, the jukebox was populated with various fractions of the requested data: 100%, 66%, 33%, and 0%. That is, in the 66% case, the jukebox is explicitly populated with 66% of the data that would be requested during the execution of the benchmark. The chunks for population of the jukebox were selected randomly. The two extreme hit-ratios of 100% and 0% give an indication of best-case performance and the system overhead. Our baseline for comparison is the execution of the benchmarks with an unmodified Coda client and server.

The recipes employed by CASPER for these experiments included a single type of recipe choice: namely, SHA-1 hashes of variable-size blocks with an average block size of 4 KB (similar to LBFS). During the experiments, none of the recipes were cached on the client. Every client cache miss generated a recipe fetch for the file. Further, because the client cache was big enough to prevent eviction of any reconstructed files, the client never requested a file or a recipe more than once.

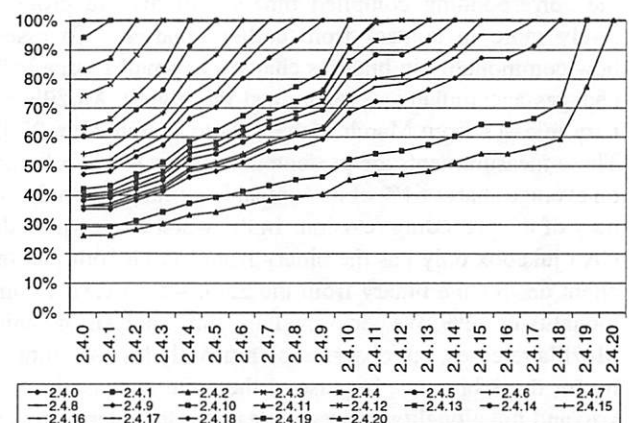
6.2 Measured Commonality

The hit rate of requests to CAS providers is highly dependent on the commonality in data between the file server and the CAS providers. Further, we expect the degree of commonality to depend on the applications that are trying to exploit it. Some applications, like the virtual machine migration benchmark used in our evaluation, would be well-suited to a CASPER approach. This benchmark reconstructs the data, including common operating system and application code, found on typical disk drives. A preliminary study by the authors of several user systems and prior work [3, 7] suggest that a high degree of commonality may be expected for this application.

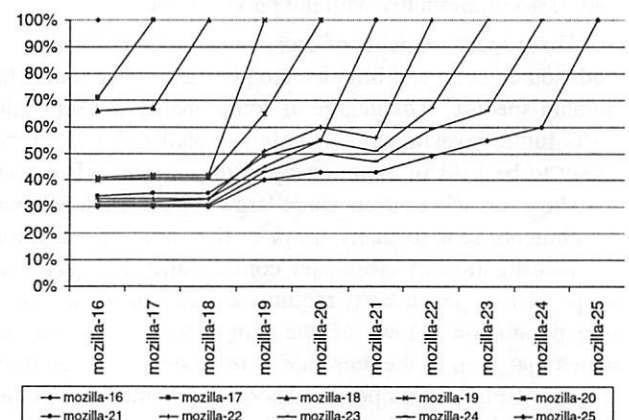
Commonality may also be expected when retrieving common binary or source-code distributions. For a software developer, different versions of a source tree also seem to exhibit this trend. For example, we compared the commonality between various releases of the Linux kernel source



(a) Commonality between Linux 2.2 Kernel releases



(b) Commonality between Linux 2.4 Kernel releases



(c) Commonality between Mozilla nightly binary releases

Each data series represents the measured commonality between a reference version of the software package and all previous releases of that package. Each point in the data series represents the percentage of data blocks from the reference version that occur in the previous release. The horizontal axis shows the set of possible previous releases, and the vertical axis relates the percentage of blocks in common. Each data series peaks at 100% when compared with itself.

Figure 4. Linux Kernel and Mozilla Commonality

code. For this study, we define commonality as the fraction of unique blocks in common between pairs of releases.

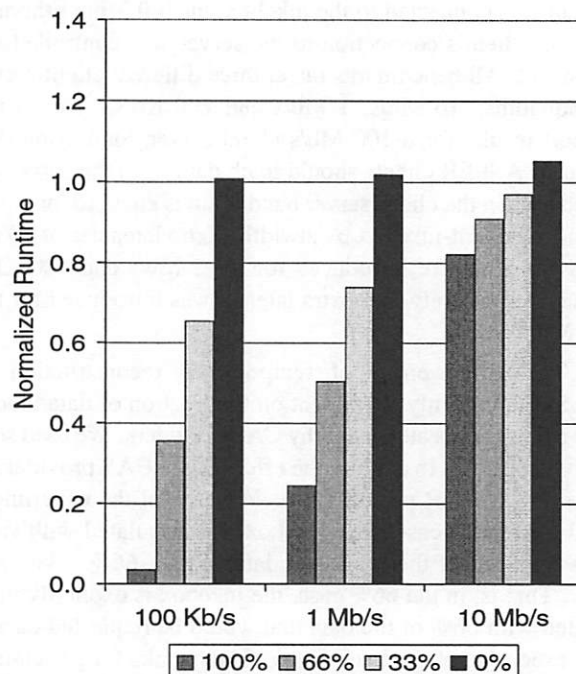
Figures 4 (a) and (b) show the measured commonality between versions of the Linux 2.2 and 2.4 kernels. We applied the block-extraction algorithm used in CASPER to each kernel's source tree and collected the unique hashes of all blocks in the tree. We then compared the set of hashes derived from each release to the set of hashes derived from each previous release in its series. The results show that commonality of 60% is not uncommon, even when the compared source trees are several versions apart. The minimal commonality we observe is approximately 30%. We will show in Section 6.3.3 that even this degree of commonality can lead to significant performance benefits from CAS in low-bandwidth conditions.

As small changes accumulate in source code over time, the corresponding compiled binary will diverge progressively more in content from earlier binaries. To assess how commonality in binaries changes as small source-level changes accumulate, we examined the nightly Mozilla binary releases from March 16th, 2003 to March 25th, 2003. These measurements are presented in Figure 4 (c). A binary on average shares 61% of its content in common with the binary of the preceding revision. In the worst case, where the CAS jukebox only has the binary from March 16th, but the client desires the binary from the 25th, we observe a commonality of 42%. We performed the same analysis on major Mozilla releases, but observed significantly less commonality for those binaries, because of the great increase in code size and functionality between releases. One interesting exception concerned release 1.2.1, a security fix; this release had 91% commonality with the previous one.

These measurements of cross-revision commonality for both source code and binaries are promising, but somewhat domain-specific. The degree of commonality in users' files bears future investigation. Highly personalized data are unlikely to be held in common by multiple users. But there may be cases where users share large objects, such as email attachments sent to many users in the same organization. An investigation of cross-user commonality is beyond the scope of this paper, as it requires a study of a substantial user population. However, we demonstrate with measurements that even in the presence of relatively little commonality, CASPER offers performance improvements to clients with a low-bandwidth connection to their file server. Moreover, our measurements show that CASPER adds very little overhead to file system operations when blocks cannot be found at a jukebox.

6.3 Benchmarks

We evaluated three different benchmarks on the CASPER file system: Mozilla software installation, an execution trace of a virtual machine application, and a modified Andrew Benchmark. The descriptions of these benchmarks together with experimental results is presented in the next three sections.



Mozilla install times with different Jukebox hit-ratios at 100 Kb/s, 1 Mb/s, and 10 Mb/s with 100 ms, 10ms and no added latency respectively. Each bar is the mean of three trials with the maximum standard deviation observed as 1%.

Figure 5. Results: Mozilla Install

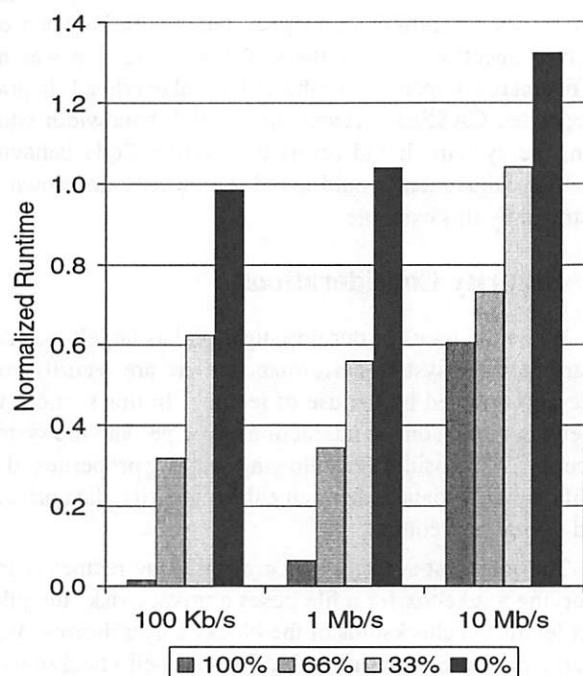
6.3.1 Mozilla Install

Software distribution is one application for which CASPER may prove effective. Often when installing or upgrading a software package, a previous version of the package may be found on the target machine or another machine near the target; the previous version may provide a wealth of matching blocks.

To evaluate software distribution using CASPER, we measured the time required to install the Mozilla 1.1 browser. The installation was packaged as 6 different RPM Package Manager [29] (RPM) files ranging from 105 KB to 10.2 MB with a total size of 13.5 MB. All the files were kept in a directory on the Coda File System and were installed by using the rpm command with the `-Uhv` options.

Figure 5 reports the time taken for the Mozilla install to complete at various network bandwidths. The times shown are the mean of three trials with a maximum standard deviation of 1%. To compare benefits at these different settings, the time shown is normalized with respect to the time taken for the baseline install with just Coda. The observed average baseline install times at 100 Kb/s, 1 Mb/s, and 10 Mb/s are 1238 seconds, 150 seconds, and 44 seconds, respectively. Note that apart from the network transfer time, the total time also includes the time taken for the rpm tool to install the packages.

As expected, the gain from using CASPER is most pronounced at low bandwidths where even relatively modest



Internet Suspend/Resume benchmark times with different Jukebox hit-ratios at 100 Kb/s, 1 Mb/s, and 10 Mb/s with 100 ms, 10ms and no added latency respectively. Each bar is the mean of three trials with the maximum standard deviation observed as 9.2%.

Figure 6. Results: ISR Benchmark

hit-rates can dramatically improve the performance of the system. For example, the reduction in benchmark execution time is 26% at 1 Mb/s with a 33% jukebox population. Further, while we do not expect the CASPER system to be used on networks with high bandwidth and low latency characteristics as in the 10 Mb/s case, the graph shows that the worst-case overhead is low (5%) and a noticeable reduction in benchmark time is possible (approximately 20% in the 100% population case).

6.3.2 Internet Suspend/Resume

Our original motivation for pursuing opportunistic use of Content Addressable Storage arose from our work on Internet Suspend/Resume (ISR) [16]. ISR enables the migration of a user's entire computing environment by layering a virtual machine system on top of a distributed file system. The key challenge in realizing ISR is the transfer of the user's virtual machine state, which may be very large. However, the virtual machine state will often include installations of popular operating systems and software packages – data which may be expected to be found on CAS providers.

To investigate the potential performance gains obtained by using CASPER for ISR, we employ a trace-based approach. For our workload, we traced the virtual disk drive accesses observed during the execution of an ISR system that was running a desktop application benchmark in demand-fetch mode. The CDA (Common Desktop Application)

benchmark uses Visual Basic scripts to automate the execution of common desktop applications in the Microsoft Office suite executing in a Windows XP environment.

By replaying the trace, we re-generate the file-access pattern of the ISR system. The ISR data of interest is stored as 256 KB files in CASPER. During trace replay, the files are fetched through CASPER on the client machine. During the benchmark, approximately 1000 such files are accessed. The trace does not include user think time.

Figure 6 shows the results from the Internet Suspend/Resume benchmark at various network bandwidths. The time shown is again the mean of three trials with a maximum standard deviation of 9.2%. The times shown are also normalized with respect to the baseline time taken (using unmodified Coda). The actual average baseline times at 100 Kb/s, 1 Mb/s, and 10 Mb/s are 5 hours and 59 minutes, 2046 seconds, and 203 seconds, respectively.

By comparing with the previous Mozilla experiment, we can see that the benefit of using CASPER is even more pronounced as the quantity of data transferred grows. Here, unlike in the Mozilla experiment, even the 10 Mb/s case shows significant gains from exploiting commonality for hit-ratios of 66% and 100%.

Note that in the ISR experiment that used a 1 Mb/s link and a 33% hit rate, CASPER reduced execution time by 44%. The interaction of application reads and writes explains this anomaly. Every time an application writes a disk block after the virtual machine resumes, the write triggers a *read* of that block, if that block has not yet been fetched from the virtualized disk. Writes are asynchronous, and are flushed to disk periodically. If CASPER speeds execution, fewer writes are flushed to disk during the shorter run time, and fewer write-triggered reads occur during that execution. Conversely, if CASPER slows execution, more writes are flushed to disk during the longer run time, and more write-triggered reads occur during that execution. This interaction between writes and reads also explains the higher-than-expected overhead in the 33% and 0% hit-ratio cases on the 10 Mb/s link.

6.3.3 Modified Andrew Benchmark

We also evaluated the system by using a modified Andrew Benchmark [13]. Our benchmark is very similar to the original but, like Pastiche [7], uses a much larger source tree. The source tree, Apache 1.3.27, consists of 977 files that have a total size of 8.62 MB. The script-driven benchmark contains five phases. Beginning with the original source tree, the scripts recreate the original directory structure, execute a bulk file copy of all source files to the new structure, stats every file in the new tree, scans through all files, and finally builds the application.

To isolate the effect of CAS acceleration on file fetch operations, the modified Andrew Benchmark experiments were executed in write disconnected mode. Thus, once the Copy

Jukebox hit-ratio	Network Bandwidth		
	100 Kb/s	1 Mb/s	10 Mb/s
100%	261.3 (0.5)	40.3 (0.9)	17.3 (1.2)
66%	520.7 (0.5)	64.0 (0.8)	20.0 (1.6)
33%	762.7 (0.5)	85.0 (0.8)	21.3 (0.5)
0%	1069.3 (1.7)	108.7 (0.5)	23.7 (0.5)
Baseline	1150.7 (0.5)	103.3 (0.5)	13.3 (0.5)

Results from the Copy Phase of the modified Andrew Benchmark run at 100 Kb/s, 1 Mb/s, and 10 Mb/s with 100 ms, 10ms and no added latency respectively. Each result is the mean of three trials with the standard deviation given in parentheses

Figure 7. Results: Modified Andrew Benchmark Copy Phase

phase was complete, a local copy of the files allowed all remaining phases to be unaffected by network conditions; only the Copy phase will show benefit from the CASPER system.

The modified Andrew benchmark differs from the Mozilla and ISR benchmarks in that the average file size is small and more than half of the files are less than 4 KB in size. As described in Section 5.2, the current system is configured such that all recipe requests for files less than 4 KB are rejected by the recipe server in favor of sending the file instead. Thus while the jukebox might have the data for that file, CASPER fetches it across the slow network link.

Note that as all operations happened in write disconnected state, MakeDir, ScanDir, ReadAll and Make were not affected by bandwidth limitations, as they were all local operations. As Figure 8 shows, these phases are not affected by network conditions in the 1 Mb/s case. We observed very similar numbers for the 100 Kb/s and 10 Mb/s cases.

Figure 7 therefore only presents results for the Copy phase for all experiments, as Copy is the only phase for which the execution time depends on bandwidth. The figure shows the relative benefits of using opportunistic CAS at different bandwidths. The time shown is the mean of three trials with standard deviations included within parentheses. Interestingly, the total time taken for the 100 Kb/s experiment with an hit-ratio of 0% is actually less than the baseline. We attribute this behavior to the fact that some of the data in the Apache source tree is replicated and consequently matches blocks in the client cache.

The results for the 10 Mb/s case show a high overhead in all cases. This benchmark illustrates a weakness of the current CASPER implementation. When many small files are transferred over the network, the performance of CASPER is dominated by sources of overhead. For example, our proxy-based approach induces several additional inter-process transfers per file relative to the baseline Coda installation. Further, the jukebox inspection introduces multiple additional network round-trip time latencies associated with the jukebox query messages and subsequent missed-block requests.

Normally, these sources of overhead would be compensated for by the improved performance of converting WAN

accesses to LAN accesses. However, for this particular benchmark the difference in peak bandwidth between our LAN connection and simulated WAN connection was not sufficient to compensate for the additional overhead. In practice, when CASPER detects such a high-bandwidth situation, the system should revert to baseline Coda behavior. Such an adjustment would avoid the system slowdown illustrated by this example.

7 Security Considerations

While the focus of our investigation has largely been on distributed file system performance, there are security considerations raised by the use of recipes. In this section, we briefly comment on the interaction of recipes and file system security. We consider the following security properties: data confidentiality, data authenticity, data integrity, data privacy, and user access control.

The foremost security concern raised by recipes is that querying a jukebox for a file poses a privacy risk: the jukebox learns the checksums of the blocks a user desires. With a large database of common blocks and their checksums, a malicious jukebox or eavesdropper could deduce the *content* a user requests. This privacy risk appears to be fundamental to our approach. An efficient solution remains an open problem.

Conventional encryption techniques also pose a challenge to recipe-based systems. If each user encrypts and decrypts her data with a distinct secret key, the same plaintext data would be encrypted into different ciphertext by users with different encryption keys. In such a case where commonality is obscured by encryption, a user could share blocks with herself alone, with marginal benefit at best. However, if we use *convergent encryption* [1, 9] (*i.e.* encrypting a block using a hash of the block as the key), recipes would be applicable—as identical data are encrypted into identical ciphertext, a recipe would again identify matching blocks.

On the other hand, recipes easily coexist with data authenticity. If data authenticity or non-repudiation is required, a file will simply contain a message authentication code or digital signature. As the authentication data are simply part of the file, recipes describe them as well. Likewise, recipes ensure data integrity. As a recipe provides strong cryptographic checksums of a file's contents, the user has a strong guarantee of a file's integrity when all its block checksums match.

A recipe-based system should enforce access control requirements for recipes that are identical to the requirements for the corresponding files. Even though recipes contain one-way hashes of file data, they still reveal information useful to a malicious user who wishes to learn the file's contents. If an attacker has an old version of a file and the up-to-date recipe for that file, he can compute many changed versions of the old file by brute force, and use checksums from the

AB phase	100%	66%	33%	0%	Baseline
MakeDir	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
Copy	40.3 (0.9)	64.0 (0.8)	85.0 (0.8)	108.7 (0.5)	103.3 (0.5)
ScanDir	2.0 (0.0)	2.0 (0.0)	2.0 (0.0)	2.3 (0.5)	2.0 (0.0)
ReadAll	3.7 (0.5)	3.7 (0.5)	4.0 (0.0)	3.7 (0.5)	3.7 (0.5)
Make	15.3 (0.5)	16.0 (0.0)	15.7 (0.5)	16.0 (0.0)	15.7 (0.5)
Total	61.3 (0.9)	85.7 (1.2)	106.7 (0.5)	130.7 (0.5)	124.7 (0.5)

Modified Andrew Benchmark run at 1 Mb/s. Each column represents the hit-ratio on the jukebox. Times are reported in seconds with standard deviations given in parentheses

Figure 8. Results: Modified Andrew Benchmark (1 Mb/s)

recipe to identify the correct version of the new file. To prevent such attacks, CASPER assigns recipes identical permissions to the files they summarize.

8 Future Extension: Fuzzy Block Matches

Our experimental results clearly show that at a given bandwidth between client and server, the dominant factor in determining the performance benefit afforded by CAS is the block hit ratio at the jukebox. This relationship suggests that strategies for increasing the hit ratio may improve the performance of CAS-enabled storage systems. Toward this end, in this section we propose *fuzzy block matching*, a promising enhancement to CAS that is the topic of our continuing research and implementation efforts.

The two recipe block types we have considered thus far, fixed- and variable-length blocks, require *exact* matches between a block's hash in a recipe and a block's hash at the jukebox. Matching hashes prove, with extremely high probability, that the jukebox and "home" file server (which generated the recipe from the canonical instance of the file data) hold the same data for a block. However, an inexact match between block contents is also possible. Here, two blocks may share many bytes in common, but differ in some bytes. Fuzzy block matching allows a CAS client to request blocks with the *approximate* contents it needs from a jukebox, and to use these approximately matching blocks to reconstruct a file so that its contents *exactly* match the canonical version of the file.

One cannot simply use as-is a block whose contents only approximately match those desired. The resulting reconstructed file would differ from the canonical file stored at the home server. After Lee *et al.* [17], we view a block at a jukebox that approximately matches the desired block as an *errored* version of the desired block, received through a noisy channel. Lee *et al.* use error-correcting codes (ECCs) to correct short replacements in similar disk blocks. We note an analogous opportunity in CAS: if we could somehow *correct* the errors in the jukebox's block, we could produce the exact desired block.

Under fuzzy block matching, the specification of a block in a recipe includes three pieces of information:

- **An exact hash value** that matches only the correct block;
- **A fuzzy hash value** that matches blocks similar to the correct block;
- **Error-correcting information** that, when applied to a block similar to the correct block, may sometimes recover the correct block.

While the hit-rate increase offered by fuzzy block matching will improve with added error-correcting information, we note that these three items must be compact in their total size. For recipes to be a viable mechanism, they must be significantly smaller than the file contents themselves. We expect that a few hundred bytes will suffice to hold useful fuzzy hash values and error-correcting information.

Fuzzy block matching works as follows:

- The CAS client sends the jukebox the enhanced recipe entry for a block, described above.
- The jukebox first determines whether it holds a block whose hash matches the exact hash value; if so, it returns this block to the client.
- The jukebox next uses the fuzzy hash value to find any *candidate blocks* it holds that approximately match the block sought by the client.
- The jukebox applies the error-correcting information to each candidate block. If the corrected block's hash matches the exact hash value from the recipe, the jukebox returns the corrected block to the client.
- If none of the jukebox's candidate blocks can be corrected to match the exact hash, the CAS client has missed in the jukebox.

There are two algorithmic aspects to fuzzy block matching: identifying approximately matching blocks with fuzzy hashing, and error-correcting approximately matching blocks, to recover correct block. We now sketch the techniques we are pursuing to solve these two problems.

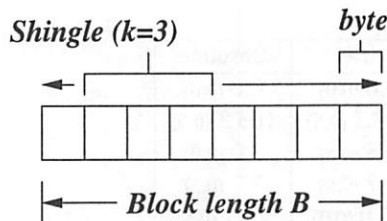


Figure 9. Structure of a disk block (length B) and shingle (k contiguous bytes). Here, $k = 3$.

8.1 Fuzzy Hashing: Shingling

The well-known technique of *shingling* [5] efficiently computes the *resemblance* of web documents to one another. Douglass and Iyengar [10] apply shingling to identify similar data objects in a file system, so that one may be delta-encoded in terms of the other. We briefly describe an adaptation of Broder *et al.*'s shingling to binary disk blocks in the CAS context.

We assume variable-length blocks, whose boundaries are determined by Rabin fingerprints over block contents, as described in Section 3. Consider a block of length B bytes. Define a *shingle* to be a sequence of k contiguous bytes in a block. Shingles overlap; there are $B - k + 1$ shingles in a block of length B bytes, one beginning at each of bytes zero through $B - k$. Figure 9 shows a shingle within a disk block. The horizontal arrows denote the “sliding” of the k -byte window within the block, to produce different shingles.

Compute a hash of each shingle in the disk block. Broder *et al.* use an enhanced version of Rabin fingerprints [25] for hashing, for reasons of computational efficiency. This procedure produces a set of integral shingle values. Sort these shingle values into numeric order, and select the m smallest unique values. These m values form a *shingleprint*, which represents the approximate contents of the disk block. Estimating the resemblance of two blocks is a simple computation involving the number of their m shingle values they share in common.

Under CAS, a shingleprint can be stored in a recipe as the fuzzy hash of a block. A client presents the shingleprint to a jukebox, which may use it to identify blocks similar to the desired block.

8.2 Using Similar Blocks: ECCs

The essence of our approach is to divide a variable-length disk block into fixed-length, non-overlapping subblocks (the last subblock may be shorter than the others), and to detect (possibly offset) subblocks that remain unchanged in content after insertions, deletions, or replacements in the full disk block. Subblocks that changed are treated as errored. By targeting a specific maximum number of changed subblocks, we can store enough ECC information to recover the original data for the whole block. In this scheme, replacements, insertions, and deletions within fixed-length blocks amount to corruptions of subblocks.

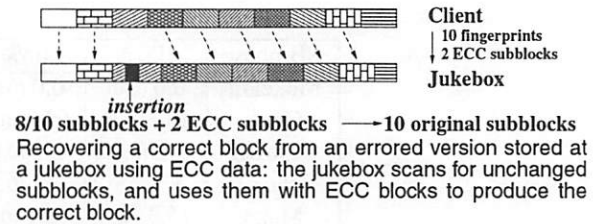


Figure 10. Example of Similar Block Correction

Again working with a disk block of length B , let us define a fixed subblock length L , where $L < B$. The client's recipe includes a Rabin fingerprint for each subblock. When the client requests similar blocks from a jukebox, it includes these Rabin fingerprints over the subblocks in the request.

For each candidate block it holds, the jukebox scans for the subblock Rabin fingerprints it receives from the client. By “scan,” we mean that the jukebox computes a Rabin fingerprint on every subblock in the candidate block, scanning each contiguous region of length L . As described previously, Rabin fingerprints are computationally efficient to compute in a sliding window at every offset in a block of data. Even if the candidate block contains insertions, deletions, and replacements, it is likely that it will contain subblocks identical to subblocks of the correct block.

An example of this ECC procedure is provided in Figure 10. Here, the client desires a disk block, subdivided into ten subblocks. The client stores in its recipe the Rabin fingerprint for each of these ten subblocks. It also stores two ECC subblocks, each 128 bytes long, originally derived from the block's canonical contents. Let us assume that the jukebox has a candidate block found by shingling, but that contains an insertion of length shorter than one subblock. The jukebox computes the Rabin fingerprint for each contiguous 128-byte subblock of the candidate block, and identifies those subblocks whose Rabin fingerprints match those provided by the client; these subblock correspondences between the correct block and jukebox block are shown by vertical dashed arrows in Figure 10. Thereafter, the jukebox applies the ECC to these unchanged subblocks and the two ECC blocks, and overcomes the “erasure” of the modified subblock, to produce the exact desired block. As a final check, the jukebox computes an exact-match hash over the whole corrected block, and compares it with the exact-match hash requested by the client. We assume here that we've chosen an ECC robust to loss of two subblocks out of ten; in practice, the degree of redundancy chosen for the ECC will reflect the degree of difference between blocks from which one wishes to recover successfully. One pays for increased robustness to changes in a candidate block with increased storage for ECC blocks in recipes.

In Figure 11, we detail the size of the four per-block components of recipes in the prototype implementation of fuzzy block matching we are currently developing. The prototype uses variable-sized blocks of mean size 4K, and subblocks 128 bytes in length. The SHA-1 exact-match hash

Content	Total Size (bytes)
SHA-1 exact hash	20
Shingleprint fuzzy hash	80
Subblock Rabin fingerprints	256
ECC information	256

Average per-block recipe storage cost in prototype supporting fuzzy block matching on variable-sized blocks (4KB mean block size).

Figure 11. Fuzzy Block Matching Recipe Storage

requires 20 bytes of storage. The shingleprint fuzzy hash consists of ten 64-bit Rabin fingerprints. The subblock fingerprints (which are also computed as Rabin fingerprints) require 64 bits each. Finally, our prototype includes two 128-byte ECC subblocks per block, and can thus successfully reconstruct a desired block from a similar block that differs in up to two 128-byte subblocks. The total recipe storage per block is 612 bytes, about 15% the size of a 4K block, but an order of magnitude longer than an exact-match SHA-1 hash alone. We note that this storage cost is conservative; fuzzy block matching with less recipe storage per block should be achievable. Truncating the Rabin fingerprints used in the shingleprint and per-subblock from 64 bits to 32 bits would reduce the storage requirement to 444 bytes, or 10.8% the size of a 4K block. Truncating these fingerprints increases the likelihood of collisions in fingerprint values, which can cause shingling to overestimate resemblance, and cause subblocks to be falsely identified as matching. Because the whole-block hash is always verified before returning a corrected block to a client, however, these collisions do not compromise the correctness of fuzzy block matching. Measuring the increase in hit rate afforded by fuzzy block matching is the subject of our continuing research.

9 Conclusion

In this paper, we have introduced the file *recipe* abstraction. File recipes are useful synopses of file contents that may be treated as first-class objects. Using this abstraction, we enhanced a distributed file system to retrieve file data from nearby Content Addressable Storage *opportunistically*. In cases where the network path between a client and its home file server offers low bandwidth, the resulting distributed file system, CASPER, offers significantly improved performance over a traditional Coda installation.

The efficiency of CASPER depends directly on the ability of the system to locate data blocks on CAS providers. Preliminary measurements of the overlap in data blocks across revisions of the Linux kernel sources and Mozilla nightly binary releases show significant commonality, even between non-consecutive pairs of releases. Finally, our ongoing work on a Fuzzy Block Matching algorithm holds promise for improving the data-block hit rate by using approximately matching blocks from CAS providers to reconstruct the exact desired contents of a file.

Acknowledgments

We would like to thank Jason Flinn and Shafeeq Sinnamohideen for their help and feedback with the design of CASPER, and Casey Helfrich for his help in setting up the test machines. We have benefited greatly from the work of Jan Harkes, Peter Braam and many other past contributors to Coda. Our shepherd, Amin Vahdat, and the anonymous reviewers gave valuable feedback and many suggestions for improving the paper. All unidentified trademarks used in this paper are properties of their respective owners.

References

- [1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI '02)* (December 2002).
- [2] BOLOSKY, W. J., CORBIN, S., GOEBEL, D., , AND DOUCEUR, J. R. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24. Seattle, WA (August 2000).
- [3] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. *ACM SIGMETRICS Performance Evaluation Review*, 28(1):34–43 (2000). ISSN 0163-5999.
- [4] BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., AND MALER, E. Extensible Markup Language (XML) 1.0 (Second Edition) (October 2000). <http://www.w3.org/TR/REC-xml>.
- [5] BRODER, A., GLASSMAN, S., MANASSE, M., AND ZWEIG, G. Syntactic Clustering of the Web. In *Proceedings of the 6th International WWW Conference* (1997).
- [6] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Lecture Notes in Computer Science*, 2009:46+ (2001).
- [7] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making Backup Cheap and Easy. In *OSDI: Symposium on Operating Systems Design and Implementation* (2002).
- [8] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*. Chateau Lake Louise, Banff, Canada (October 2001).
- [9] DOUCEUR, J. R., ADYA, A., BOLOSKY, W. J., SIMON, D., AND THEIMER, M. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS 2002)* (July 2002).
- [10] DOUGLIS, F. AND IYENGAR, A. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of the USENIX Annual Technical Conference*. San Antonio, Texas (June 2003).
- [11] DRUSCHEL, P. AND ROWSTRON, A. PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In *HotOS VIII*, pages 75–80. Schloss Elmau, Germany (May 2001).
- [12] FLINN, J., SINNAMOHIDEEN, S., TOLIA, N., AND SATYANARAYANAN, M. Data Staging on Untrusted Surrogates. In *Proceedings of the FAST 2003 Conference on File and Storage Technologies* (2003).
- [13] HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1) (February 1988).
- [14] HOWES, T. A. AND SMITH, M. C. A Scalable, Deployable, Directory Service Framework for the Internet. Technical report, Center for Information Technology Integration, University of Michigan (1995).

- [15] KIM, M., COX, L. P., AND NOBLE, B. D. Safety, Visibility and Performance in a Wide-Area File System. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*. Monterey, CA (January 2002).
- [16] KOZUCH, M. AND SATYANARAYANAN, M. Internet Suspend/Resume. In *Fourth IEEE Workshop on Mobile Computing Systems and Applications*. Callicoon, New York (June 2002).
- [17] LEE, Y.-W., LEUNG, K.-S., AND SATYANARAYANAN, M. Operation-based Update Propagation in a Mobile File System. In *Proceedings of the USENIX Annual Technical Conference*. Monterey, California (June 1999).
- [18] MOGUL, J. C., DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *SIGCOMM*, pages 181–194 (1997).
- [19] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. Chateau Lake Louise, Banff, Canada (October 2001).
- [20] MUTHITACHAROEN, A., MORRIS, R., GIL, T., AND CHEN, B. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*. Boston, Massachusetts (December 2002).
- [21] NAGLE, J. RFC 896: Congestion Control in IP/TCP Internetworks (January 1984).
- [22] NIST. Secure Hash Standard (SHS). In *FIPS Publication 180-1* (1995).
- [23] NIST Net. <http://snad.ncsl.nist.gov/itg/nistnet/>.
- [24] QUINLAN, S. AND DORWARD, S. Venti: A New Approach to Archival Storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies* (2002).
- [25] RABIN, M. Fingerprinting by Random Polynomials. In *Harvard University Center for Research in Computing Technology Technical Report TR-15-81* (1981).
- [26] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A Scalable Content Addressable Network. In *Proceedings of ACM SIGCOMM 2001* (2001).
- [27] RIVEST, R. The MD5 Message-Digest Algorithm. *RFC 1321* (1992).
- [28] ROWSTRON, A. AND DRUSCHEL, P. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. Heidelberg, Germany (November 2001).
- [29] RPM Software Packaging Tool. <http://www.rpm.org/>.
- [30] SAITO, Y., KARAMANOLIS, C., KARLSSON, M., AND MAHALINGAM, M. Taming Aggressive Replication in the Pangaea Wide-Area File System. In *OSDI: Symposium on Operating Systems Design and Implementation* (2002).
- [31] SATYANARAYANAN, M., EBLING, M. R., RAIFF, J., BRAAM, P. J., AND HARKES, J. *Coda File System User and System Administrators Manual*. Carnegie Mellon University (1995).
- [32] SATYANARAYANAN, M., KISTLER, J., KUMAR, P., M.E., O., SIEGEL, E., AND STEERE, D. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4) (April 1990).
- [33] SATYANARAYANAN, M. The Evolution of Coda. *ACM Transactions on Computer Systems*, 20(2) (May 2002).
- [34] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M.F., BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001*. San Diego, CA (August 2001).
- [35] SYSINTERNALS. <http://www.sysinternals.com>.
- [36] TICHY, W. F. RCS - A System for Version Control. *Software - Practice and Experience*, 15(7):637–654 (1985).
- [37] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. Ph.D. thesis, The Australian National University (1999).
- [38] Universal Plug and Play. <http://www.upnp.org/>.

System Support for Online Reconfiguration

Craig A. N. Soules[†] Jonathan Appavoo[‡] Kevin Hui[‡] Robert W. Wisniewski[§]
Dilma Da Silva[§] Gregory R. Ganger[†] Orran Krieger[§] Michael Stumm[‡]
Marc Auslander[§] Michal Ostrowski[§] Bryan Rosenberg[§] Jimi Xenidis[§]

Abstract

Online reconfiguration provides a way to extend and replace active operating system components. This provides administrators, developers, applications, and the system itself with a way to update code, adapt to changing workloads, pinpoint performance problems, and perform a variety of other tasks while the system is running. With generic support for interposition and hot-swapping, a system allows active components to be wrapped with additional functionality or replaced with different implementations that have the same interfaces. This paper describes support for online reconfiguration in the K42 operating system and our initial experiences using it. It describes four base capabilities that are combined to implement generic support for interposition and hot-swapping. As examples of its utility, the paper describes some performance enhancements that have been achieved with K42's online reconfiguration mechanisms including adaptive algorithms, common case optimizations, and workload specific specializations.

1 Introduction

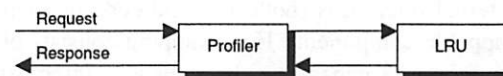
Operating systems are big and complex. They are expected to serve many needs and work well under many workloads. Often they are meant to be portable and work well with varied hardware resources. As one would expect, this demanding set of requirements is difficult to satisfy. As a result, patches, updates, and enhancements are common. In addition, tuning activities (whether automated or human-driven) often involve dynamically adding monitoring capabilities and then reconfiguring the system to better match the specific environment.

Common to all of the above requirements is a need to modify system software after it has been deployed. In some cases, shutting down the system, updating the software, and restarting is sufficient. But, this approach comes with a cost in system availability and (often) human administrative time. Also, restarting the system to add monitoring generally clears the state of the system. This can render the new monitoring code ineffective un-

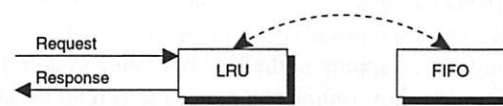
[†] Carnegie Mellon University

[‡] University of Toronto

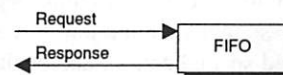
[§] IBM T. J. Watson Research Center



(a) After profiler is interposed around the page manager.



(b) LRU page manager before hot-swap with FIFO.



(c) After FIFO page manager is fully swapped.

Figure 1: Online reconfiguration. This figure shows two online reconfiguration mechanisms: interposition and hot-swapping. (a) shows an LRU page manager and an interposed profiler that can watch the component's calls/returns to see how it is performing. If it determines that performance is poor, it may decide to switch the existing component for another. (b) shows the LRU page manager as it is about to be swapped with a FIFO page manager. Once the swap is complete, as shown in (c), the LRU page manager can be deleted.

less the old state of the system can be reproduced.

Online reconfiguration can provide a useful foundation for enhancement of deployed operating systems. With generic support built into an OS's core, activities that require system modification could be addressed without adding new complexity to individual subsystems. For example, patches and updates could be applied to the running system, avoiding down-time and associated human involvement. In addition, monitoring code could be dynamically added and removed, gathering system measurements when desired without common-case overhead.

Figure 1 illustrates two basic mechanisms for online reconfiguration: interposition and hot-swapping. Interposition wraps an active component, extending its functionality with wrapper code that executes before and after each call to the component. Hot-swapping replaces an active component with a new implementation. Both modify an active component while maintaining availability of the component's functionality.

To implement interposition and hot-swapping are implemented using four capabilities. First, the system must be able to identify and encapsulate the code and data for a swappable component. Second, the system must be able to quiesce a swappable component such that no external references are actively using its code and data. Third, the system must be able to transfer the internal state of a swappable component to a replacement version of that component. Fourth, the system must be able to modify all external references (both data and code pointers) to a swappable component. Hot-swapping consists of exercising the four capabilities in sequence. Interposition relies mainly on the fourth capability to redirect external references to the interposed code.

This paper discusses these four capabilities in detail, describes our initial prototype implementation of interposition and hot-swapping in the K42 operating system [32], and discusses how online reconfiguration could be added to more traditional operating systems. To explore some of the flexibility and value of online reconfiguration, we implemented a number of well-known dynamic performance enhancements. These include common-case specialization, workload-based specialization, and scalability specialization.

2 Why Online Reconfiguration

There are a variety of reasons for modifying a deployed operating system. The most common examples are component upgrades, particularly patches that fix discovered security holes. Other examples include dynamic monitoring, system specializations, adaptive performance enhancements, and integration of third-party modules. Usually, when they are supported, distinct case-by-case mechanisms are used for each example.

This section motivates integrating into system software a generic infrastructure for extending and replacing active system components. First, it discusses two aspects of online reconfiguration: interposition and hot-swapping. Then, it discusses a number of common OS improvements, and how interposition and hot-swapping can simplify and enhance their implementation.

2.1 Online reconfiguration

Online reconfiguration support can simplify the dynamic updates and changes wanted for many classes of system enhancement. Thus, a generic infrastructure can avoid a collection of similar reconfiguration mechanisms. Two mechanisms that provide such a generic infrastructure are interposition and hot-swapping.

Interposition wraps an active component's interface, extending its functionality. Interposition wrappers may be

specific to a particular component or generic enough to wrap any component. For example, a generic wrapper might measure the average time threads spend in a given component. A component-specific wrapper for the fault handler might count page faults and determine when some threshold of sequential page faults has been reached.

Hot-swapping replaces an active component with a new component instance that provides the same interface and functionality. To maintain availability and correctness of the service provided, the new component picks up where the old one left off. Any internal state from the old component is transferred to the new, and any external references are relinked. Thus, hot-swapping allows component replacement without disrupting the rest of the system and does not place additional requirements on the clients of the component.

2.2 Applying online reconfiguration

Interposition and hot-swapping are general tools that can provide a foundation for dynamic OS improvement. The remainder of this section discusses how some common OS enhancements map onto them.

Patches and updates: As security holes, bugs, and performance anomalies are identified and fixed, deployed systems must be repaired. With hot-swapping, a patch can be applied to a system immediately without the need for down-time (scheduled or otherwise). This capability avoids a trade-off among availability and correctness, security, and better performance.

Adaptive algorithms: For many OS resources, different algorithms perform better or worse under different conditions. Adaptive algorithms are designed to combine the best attributes of different algorithms by monitoring when a particular algorithm would be best and using the correct algorithm at the correct time. Using online reconfiguration, developers can create adaptive algorithms in a modular fashion, using several separate components. Although in some cases implementing such an adaptive algorithm may be simple, this approach allows adaptive algorithms to be updated and expanded while the system is running. Each independent algorithm can be developed as a separate component, hot-swapped in when appropriate. Also, interposed code can perform the monitoring, allowing easy upgrades to the monitoring methodology and paying performance penalties only during sampling. Section 6.3 evaluates using online reconfiguration to provide adaptive page replacement.

Specializing the common case: For many individual algorithms, the common code path is simple and can be implemented efficiently. However, supporting all of the complex, uncommon cases often makes the implementa-

tion difficult. To handle these cases, a system with online reconfiguration can hot-swap between a component specialized for the common case and the standard component that handles all cases. Another way of getting this behavior is with an IF statement at the top of a component with both implementations. A hot-swapping approach separates the two implementations, simplifying testing by reducing internal states and increasing performance by reducing negative cache effects of the uncommon case code [40]. Section 6.3 evaluates the use of online reconfiguration to specialize exclusive access to a file, while still supporting full sharing semantics when necessary.

Dynamic monitoring: Instrumentation gives developers and administrators useful information in the face of system anomalies, but introduces overheads that are unnecessary during normal operation. To reduce this overhead, systems provide “dynamic” monitoring using knobs to turn instrumentation on and off. Interposition allows monitoring and profiling instrumentation to be added when and where it is needed, and removed when unnecessary. In addition to reducing overhead in normal operation, interposition removes the need for developers to guess where probes would be useful ahead of time. Further, many probes are generic (e.g., timing each function call, counting the number of parallel requests to a component). Such probes can be implemented once, avoiding code replication across components.

Application-specific optimizations: Application specializations are a well-known way of improving a particular application’s performance based on knowledge only held by the application [18, 20, 22, 53]. Using online reconfiguration, an application can provide a new specialized component and swap it with the existing component implementation. This allows applications to optimize any component in the system without requiring system developers to add explicit hooks to replace each one.

Third-party modules: An increasingly common form of online reconfiguration is loadable kernel modules. Particularly with open-source OSes, such as Linux, it is common to download modules from the web to provide functionality for specialized hardware components. In the case of Linux, the module concept also has a business benefit, because a dynamically loaded module is not affected by the GNU Public License. As businesses produce value-adding kernel modules (such as “hardened” security modules [28, 48]), the Linux module interface may evolve from its initial focus on supporting device drivers toward providing a general API for hot-swapping of code in Linux. The mechanisms described in this paper are a natural endpoint of this evolution, and the transition has begun; we have worked with Linux developers to implement a kernel module removal scheme using

quiescence [36].

2.3 Summary

Online reconfiguration is a powerful tool that can provide a number of useful benefits to developers, administrators, applications, and the system itself. Each individual example can be implemented in other ways. However, generic support for interposition and hot-swapping can support them all with a single infrastructure. By integrating this infrastructure into the core of an OS, one makes it much more amenable to subsequent change.

3 Online Reconfiguration Support

This section discusses four main requirements of online reconfiguration. First, components must have well-defined interfaces that encapsulate their functionality and data. Second, it must be possible to force an active component into a quiescent state long enough to complete state transfer. Third, there must be a way to transfer the state of an existing component to a new component instance. Fourth, it must be possible to update external references to a component.

3.1 Component boundaries

Each system component must be self-contained with a well-defined interface and functionality. Without clear component boundaries, it is not possible to be sure that a component is completely interposed or swapped. For example, an interposed wrapper that counts active calls within a component would not notice calls to unknown interfaces. Similarly, any component that stores its state externally cannot be safely swapped, because any untransferred external data would likely lead to improper or unpredictable behavior.

Achieving clear component boundaries requires some programming discipline and code modularity. Using an object-oriented language can help. Components can be implemented as objects, encapsulating functionality and data behind a well-defined interface. Object boundaries help prevent confusing code and data sharing, often resulting in cleaner components and a more maintainable code base. Although it may be possible to detect some rule violations using code analysis [17], any solution still relies on developer diligence and adherence to the programming discipline.

3.2 Quiescent States

Before a component can be swapped, the system must ensure that all active use of the state of the component has concluded. Without such quiescence, active calls

could change state while it is being transferred, causing unpredictable behavior.

Operating systems are often characterized as event driven. The majority of activity in the OS can be represented and serviced as individual requests with any given request having an identifiable start and end. This nature can be leveraged to implement a number of interesting synchronization algorithms. By associating idempotent changes of system data structures with an epoch of requests, one can identify states in which a data structure is no longer being referenced. For example, to swing the head pointer of a linked list from one chain of nodes to another, one can divide the accesses to the list into two epochs. The first epoch includes all threads in the system that were active before the swing, and the second epoch includes any new threads begun after the swing. Because new threads will only be able to access the new chain of nodes, nodes of the old chain are guaranteed to no longer be in use once the threads in the first epoch have ended. At this point, the old chain is quiescent and can be modified at will (including being deleted).

We have utilized this style of synchronization, termed *Read-Copy-Update(RCU)*, to implement a semi-automatic garbage collector [21] and hot-swapping in K42. Others have used it in PTX [37] and in Linux to implement a number of optimizations such as: lock-free module loading and unloading [36, 44, 45].

The key to leveraging *RCU* techniques is being able to divide the work of the system into short-lived “requests” that have a cheaply identifiable start and end. In non-preemptive systems such as PTX and Linux¹, a number of key points (e.g., system calls and context switches) can be used to identify the start and end of requests. K42 is preemptable and has been designed for the general use of *RCU* techniques by ensuring that all system requests are handled on short-lived system threads. As such, thread creation and termination can be used to identify the start and end of requests. Section 4 describes how K42’s thread generation mechanism is used to determine quiescent states.

3.3 State transfer

State transfer synchronizes the state of a new component instance with that of an existing component. To complete a successful state transfer, all of the information required for proper component functionality must be packaged, transferred, and unpackaged. This requires that the old and new component agree upon both a package format and a transfer mechanism.

¹Schemes for extending the current *RCU* support in Linux to preemptive versions have been proposed[36].

There is not likely to be a single “catch-all” solution; both the data set and data usage can vary from component to component. Instead, there is likely to be a variety of packaging and transfer mechanisms suited to each component. While it is impossible to predict what all of these mechanisms will be, there are likely to be a few common ones. For example, an upgraded component will often understand the existing component in detail. Transferring a reference to the old component should be sufficient for the new component to extract the necessary state.

Given that no single mechanism exists for state transfer, a system can still provide two support mechanisms to simplify its implementation. First, the hot-swapping mechanism should provide a negotiation protocol that helps components decide upon the most efficient transfer mechanisms understood by both. Second, components that share a common interface and functionality should understand (at the least) a single, canonical data format. By ensuring this, component developers need only implement two state transfer functions to have a working implementation: to and from the canonical form.

3.4 External references

Whenever a component is interposed or hot-swapped, its external interface is redirected to a new piece of code (the wrapper for interposition, the new component for hot-swapping). Because all calls must be routed through this new code, all external references to the original component must be updated.

Reference counting and indirection are two common ways to handle this. Reference counting, as is used in garbage collection [6, 12, 26], tracks all references to a component (even within a single client). If a component changes, all tracked references are found and updated. The drawback of reference counting is that its overhead grows linearly with the number of component references. On the other hand, indirection requires that all references point to a single indirection pointer. To update all of a client’s references to an object, the system only needs to update the single indirection pointer for that client. Similarly, if a globally accessible object is being swapped, the indirection pointer of each client using the object must be updated. This is space and performance efficient, with small constant overhead per component used in each client. The drawback is that the programmer must be aware of the indirection and account for it, while reference counting is handled transparently.

3.5 Orthogonal safety issues

The focus of this work is on the mechanics of online re-configuration. There are orthogonal issues of safety and

security related to deciding which reconfigurations to permit [7, 41] and containing suspect extensions [50, 52]. Other researchers have addressed these issues with several different proposals. The implementation described in Section 4 makes no guarantees about the safety of a reconfiguration. However, most prior techniques for ensuring safety (see Section 7) could be applied to this implementation.

4 Online Reconfiguration in K42

This section describes the integration of online reconfiguration into the K42 operating system. It overviews K42, describes the features used, and details the implementations of interposition and hot-swapping.

4.1 K42

K42 is an open-source research OS for cache-coherent 64-bit multiprocessor systems. It uses an object-oriented design to achieve good performance, scalability, customizability, and maintainability. K42 supports the Linux API and ABI [3] allowing it to use unmodified Linux applications and libraries. The system is fully functional for 64-bit applications, and can run codes ranging from scientific applications to complex benchmarks like SDET to significant subsystems like Apache.

In K42, each virtual resource instance (e.g., a particular file, open file instance, memory region) is implemented by combining a set of (C++) objects [5]. For example, there is no global page cache in K42; instead, for each file, there is an independent object that caches the blocks of that file. While we believe that online reconfiguration is useful in general systems, the object-oriented nature of K42 makes it a particularly good platform for exploring fine-grained hot-swapping and interposition.

4.2 Support mechanisms

The four requirements of online reconfiguration are addressed as follows.

Component boundaries: K42's object-oriented approach naturally maps each system component onto a C++ language object. K42 requires that the external interface of a component is defined as a virtual base class for all implementations of the component. This programmer convention enforces the required component boundaries without significant burden on developers.

Quiescent States: K42 employs a technique similar to that discussed in Section 3 to establish quiescent states for an object. In K42, all system requests are serviced by a new system thread. Hence, requests can be divided into epochs by partitioning the threads. Specifically, it is pos-

sible to determine when all threads that were in existence on a processor at a specific instance in time have terminated. K42 maintains two thread generations to which threads are assigned.² Each generation records the number of threads that are active and assigned to it. At any given time, one of the generations is identified as the current generation and all new threads are assigned to it. To determine when all the current threads have terminated, the following algorithm is used:

```
i := 0
while (i < 2)
  if (non-current generation's count = 0)
    make it the current generation
  else
    wait until it is zero and
    make it the current generation
  i := i+1
```

In K42, the process of switching the current generation is called a generation swap. The above algorithm illustrates that two generation swaps are required to establish that the current set of threads have terminated. This mechanism is timely and accurate even in the face of preemption, because K42's design does not use long-lived system threads nor does it rely on blocking system-level threads. Note that in the actual implementation, the wait is implemented via a call-back mechanism, avoiding a busy wait.

State transfer: K42 leaves the implementation of individual state transfer methods up to the developer. However, to assist state transfer negotiation, K42's online reconfiguration mechanism provides a *transfer negotiation protocol*. For each set of functionally compatible components, there must be a set of state transfer protocols that form the union of all possible state transfers between these components. For each component, the developers must create a prioritized list of the state transfer protocols that it supports. For example, it may be best to pass internal structures by memory reference, rather than copying the entire structure; however, both components must understand the same structure for this to be possible. Before initiating a hot-swap, K42 requests these two lists from the old and new component instances. After determining the most desirable format based on the two lists, K42 requests the correct package format from the old component and passes it to the new component. Once the new component has unpackaged the data, the transfer is complete.

External references: K42 uses the per-client *object translation table* to provide a layer of indirection for accessing system components. When an object instance is

²The design supports an arbitrary number of generations but only two are used currently.

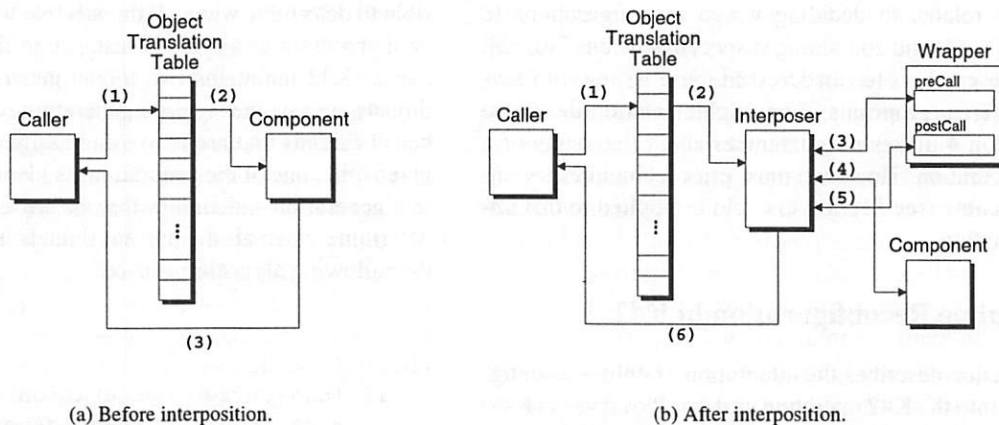


Figure 2: **Component interposition.** This figure shows the steps of component interposition. (a) shows how callers access components through the object translation table. In this case, calls from the caller lookup the component in the object translation table, and then call the component based on that indirection. (b) shows an interposed component. In this case, the caller's indirection points the call at the generic interposer. The interposer then makes three calls, first to the wrapper's PRECALL, then the original component call, then the wrapper's POSTCALL.

created, an entry for it is created in the object translation table of clients that are accessing it, and all external calls to the component are made through this reference. K42 can perform a hot-swap or interposition on this component by updating the entry in the appropriate tables. Although this incurs an extra pointer dereference per component call, the object translation table has other benefits (e.g., improved SMP scalability [21]) that outweigh this overhead.

4.3 Online reconfiguration

The remainder of this section describes how K42 utilizes the four system support features described above to provide interposition and hot-swapping.

4.3.1 Interposition

Object interposition interposes additional functionality around all function calls to an existing object instance. We partition interposed functionality into two pieces: a *wrapper object* and a *generic interposer*. The wrapper object contains the specific code that should be executed before and/or after a call to an object. The generic interposer wraps the interface of any object, transparently calling the functions of a given wrapper object before forwarding the call to the original object.

The wrapper object is a standard C++ object with two calls: PRECALL and POSTCALL. As one might suspect, the former is called before the original object's function, and the latter is called after. In these calls, a wrapper can maintain state about each function call that is in flight, collect statistical information, modify call parameters and return values, and so on.

To redirect calls from the original object to the generic interposer, the interposer replaces itself for the object in the object translation table. To handle arbitrary object interfaces, K42's interposer leverages the fact that every external call goes through a virtual function table. Once the generic interposer replaces the original object in the object translation table, all calls go through the interposer's virtual function table. The interposer overloads the method pointers in its virtual function table to point at a single *interposition method*, forcing all external calls through this function. The interposition method handles calls to the wrapper object's methods as well as calling the appropriate method of the original component, as shown in Figure 2. It also determines which of the original methods was called, allowing method specific wrappers.

To handle arbitrary call parameters and return values, the interposer method must ensure that all register and stack state is left untouched before the call is forwarded to the original component. At odds with this requirement is the need to store information normally kept on the stack (e.g., the original return address, local variables). To resolve this conflict, the interposer allocates space for any required information on the heap and keeps a pointer to it in a callee-saved register. This register's value is guaranteed to be preserved across function calls; when control is returned to the interposer, it can retrieve any saved information. The information saved in the heap space must include both the original return address and the original value of the callee-saved register being used, because it must be saved by the callee for the original caller.

The division of labor between the generic interposer and the wrapper object was chosen because the interposition method can not make calls to the generic inter-

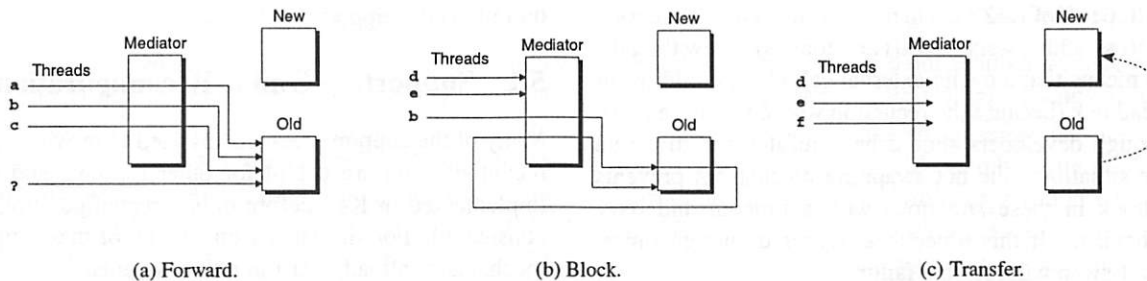


Figure 3: Component hot-swapping. This figure shows the three phases of hot-swapping: forward, block, and transfer. In the forward phase, new calls are tracked and forwarded while the system waits for untracked calls to complete. Although this phase must wait for all old threads in the system to complete, all threads are allowed to make forward progress. In the block phase, new calls are blocked while the system waits for the tracked calls to complete. By blocking only tracked calls into the component, this phase minimizes the blocking time. In the transfer phase, all calls to the component have been blocked, and state transfer can take place. Once the transfer is complete, the blocked threads can proceed to the new component and the old component can be garbage collected.

poser's virtual function table. If the wrapper and interposer were combined (using a parent interposer, and inheriting for each specific case), then the interposition method would have to locate the correct PRECALL and POSTCALL from the specialized interposer's internal interfaces. Unfortunately, this is difficult, because they would be in compiler-specified locations that are difficult to determine at run-time. By separating the wrapper, we can avoid specializing the interposition method for each wrapper, because the PRECALL and POSTCALL can be located using the wrapper's virtual function table.

To detach an interposed wrapper, the corresponding interposer object replaces its object translation table entry with a pointer to the original object. Once the object translation table is updated, all incoming calls will be sent directly to the original object. Garbage collection of the interposer and wrapper can happen out-of-band, once they quiesce.

4.3.2 Hot-swapping

K42's object hot-swapping mechanism builds on interposition. The first step of hot-swapping from the current object instance (X) to a new object instance (Y) is to interpose X with a *mediator*. Before the mediator can swap objects, it must ensure that there are no in-flight calls to X (i.e., the component must be in a quiescent state). To get to this state, the mediator goes through a three phase process: forward, block, and transfer.

In the *forward* phase, the mediator tracks all threads making calls to the component and forwards each call to the original component. This phase continues until all calls started before call tracking began have completed. To detect this, K42 relies on the thread generation mechanism. When the forward phase begins, a request to establish a quiescent state with respect to the current

threads is made to the generation mechanism. After two generation swaps, the generation mechanism calls back, because all current threads are guaranteed to have terminated.

Figure 3a illustrates the forward phase. When the mediator is interposed, there may already be calls in progress; in this example, there is one such call marked as ?. While waiting for a quiescent state, new calls **a**, **b**, and **c** are tracked by the mediator. The next phase begins once the generation mechanism indicates that a quiescent state has been achieved.

The mediator begins the *blocked* phase once all calls in the component are tracked. In this phase, the mediator temporarily blocks all new incoming calls, while it waits for the calls it has been tracking to complete. An exception must be made for incoming recursive calls, because blocking them would create deadlock. Once all the tracked calls have completed, the component is in a quiescent state, and the state transfer can begin.

Figure 3b illustrates the blocked phase. In this case, thread **b** is in progress, and must complete before the phase can complete. New calls **d** and **e** are blocked; however, because blocking **b** during its recursive call would create deadlock, it is allowed to continue.

Although it would be simpler to combine the forward and blocked phase (blocking new calls immediately and waiting for then generation mechanism to indicate a quiescent state), this would require blocking until all threads in the system have completed. Due to K42's event driven model, thread lifetimes are guaranteed to be short; however, blocking an overloaded component while waiting for every thread in the system to expire may reduce component availability and overall system performance. By separating the phases, blocking is only dependent upon the lifetime of threads active in that component.

One tradeoff of K42's event model is that cross-processor and cross-address-space calls are done with new threads. This means that a cyclic external call chain could result in deadlock (because the recursion would not be caught). Although developers should be careful never to create these situations, the hot-swapping mechanism prevents deadlock in these situations with a timeout and retry mechanism. If this timeout is triggered enough times, the hot-swap will return a failure.

After the component has entered a quiescent state, the mediator begins the *transfer* phase. In this phase, the mediator performs state transfer between the old and new components, updates the object translation table entry to point at the new component, and then allows blocked calls to continue to the new component. Each set of functionally compatible components share a set of up to 64 state transfer protocols. Acceptable protocols are specified using a bit vector that is returned from each component. The intersection of these vectors gives the list of potential protocols. The mediator determines the best common format by requesting the protocol vector from each component, and then choosing the protocol corresponding to the highest common bit in the vector. Once a protocol is decided upon, the mediator retrieves the packaged state from the original component and passes it to the new component.

Figure 3c illustrates the transfer phase. Once the old component is quiescent, all state is transferred to the new component. Once the unpacking completes, threads **d**, **e**, and **f** are unblocked and sent to the new component. At this point, the mediator is detached and the old component destroyed.

4.4 Summary

K42's online reconfiguration mechanisms handle call interception and mediation transparently to clients with external interfaces, and separates the complexities of swap-time, in-flight call tracking and deadlock avoidance from the implementation of the component itself. With the exception of component state transfer, the online reconfiguration process does not require support from the component, simplifying the creation of components that wish to take advantage of interposition or hot-swapping. Although not described, considerable effort has gone into ensuring that the interposition and hot-swapping mechanisms are scalable and efficient on a multiprocessor; see [4] for details.

5 Support in Other Systems

This section discusses generally how online reconfiguration can be supported in systems other than K42 and specifically the changes and additions that would be re-

quired to add support into Linux.

5.1 Supporting Online Reconfiguration

Many of the support mechanisms used to provide online reconfiguration are useful for other reasons, and were implemented in K42 before online reconfiguration was considered. For similar reasons, many of these support mechanisms already exist in many systems.

As is discussed in Section 3, an object-oriented design leads to better code structure, cleaner interfaces, more maintainable code, and a more modular approach that improves scalability. For this reason, modular approaches and object-oriented designs are becoming increasingly common in operating systems (e.g., the VFS layer in UNIX, shared libraries, plug-and-play device drivers, loadable module support). As systems incrementally add this modularity, more and more components in the system can become eligible for online reconfiguration.

PTX and Linux already have explicit support for RCU techniques, as described in Section 3. Hence, establishing the quiescent state required for hot swapping is straightforward. RCU support can be utilized to implement a number of lock-free synchronization algorithms beyond just hot-swapping. In PTX and Linux RCU, support was added to facilitate several independent lock-free optimizations. But, perhaps more importantly, the use of RCU in PTX and Linux illustrates the fact that the RCU approach used in K42 for hot-swapping generalizes and is equally applicable in traditional systems. This is not surprising, because the event-driven nature of operating systems is widely accepted and is the fundamental property on which RCU relies.

Because indirection leads to added flexibility, most systems have several points of indirection built-in. For example, the indirection used in the VFS layer of many operating systems abstracts the underlying file system implementation from the rest of the system, isolating the other components of the system from the internals of the various file systems. Other examples of indirection include device drivers and virtual memory systems.

In K42, rather than having many different styles or points of indirection, a standard level of indirection was introduced in front of all objects. Directing every object access through an indirection allows K42 to support complex multiprocessor optimizations while preserving a simple object-oriented model [21]. However, it also has allowed us to entertain hot-swapping any instance of an object. Although existing systems might not have such a uniform model of indirection, hot-swapping could first be applied to the components which do live behind a level of indirection, such as VFS modules. If the flex-

ibility of an indirection proves to be useful, one would expect more and more components to utilize it. In time, perhaps standard support for accessing all components behind a level of indirection, similar to K42, would be employed.

Although the state transfer protocol has no clear additional benefits beyond online reconfiguration, this mechanism is not attached to any particular design of the system. Adding this final support mechanism to any system that wished to take advantage of online reconfiguration should be as straightforward as adding it to K42 was.

5.2 Online Reconfiguration in Linux

This section examines each of the four requirements for online reconfiguration and discusses how to use existing mechanisms and add additional mechanisms to Linux to support online reconfiguration.

Component boundaries: Although Linux is not strictly modular in design, interface abstractions have been added to support loadable modules in many places. These well-defined abstractions could be used to provide the component boundaries required by online reconfiguration.

Quiescent States: Over the last few years, a number of patches have been developed for Linux [36] to add support for RCU mechanisms. This support has been leveraged for a number of optimizations [44] including: adding lock-free lookups of the *dentry* cache, supporting hot-plugging of CPUs, safe module loading and unloading, scalable file descriptor management, and lock-free lookups in the IPV4 route cache. The Linux 2.5 kernel has recently integrated this support into the main line version [45]. This same RCU infrastructure can be utilized to identify the necessary quiescent state for hot-swapping, as is done in K42.

State transfer: Although the boundaries for module state are not as well-defined in Linux as they are in K42, state transfer should be very similar. Also, the same transfer negotiation protocol used by K42 could be applied to Linux.

External references: In Linux, the same module abstraction used to provide component boundaries could also be used to handle external references. Because modules would lie behind a virtual interface, updating the pointers behind this interface is all that is required to replace a given module. For example, replacing a file system module (or individual pieces of file system functionality) could be done by replacing the appropriate function pointers for that file system in the VFS layer.

6 Evaluation

In this section, we evaluate the flexibility and performance of K42's online reconfiguration. First, we quantify the overheads and latencies of interposition and hot-swapping. Second, we illustrate the flexibility of online reconfiguration by using it to implement a number of well-known dynamic performance enhancements.

6.1 Experimental setup

The experiments were run on two different machines. Both of them were RS/6000 IBM PowerPC bus-based cache-coherent multiprocessors. One was an S85 Enterprise Server with 24 600MHZ RS64-IV processors and 16GB of main memory. The other was a 270 Workstation with 4 375MHZ Power3 processors and 512MB of main memory. Unless otherwise specified, all results are from the S85 Enterprise Server.

Throughout the evaluation, we use two separate benchmarks: Postmark and SDET.

Postmark was designed to model a combination of electronic mail, netnews, and web-based commerce transactions [33]. It creates a large number of small, randomly-sized files and performs a specified number of transactions on them. Each transaction consists of a randomly chosen pairing of file creation or deletion with file read or append. All random biases, the number of files and transactions, and the file size range can be specified via parameter settings. Unless otherwise specified, we used 1000 files, 10,000 transactions, file sizes ranging from 128B to 8KB, and even biases.

SDET executes one or more scripts of user commands designed to emulate a typical software-development environment (e.g., editing, compiling, and various UNIX utilities). The scripts are generated from a predetermined mix of commands,³ and are all executed concurrently. It makes extensive use of the file system and memory-management subsystems, making it useful for scalability benchmarking. Throughout this section we will refer to an "N-way SDET" which describes running N concurrent scripts on a machine configured with N processors.

6.2 Basic overheads

During normal operation, the only overhead of online reconfiguration is the indirection used to update external references. In K42, this is done using the object translation table and C++ virtual function table. The overhead of the object translation table is a single memory load, and this data is likely to be cached for frequently

³We do not run the compile, assembly and link phases of SDET, because, at the time of this paper, gcc executing on a 64-bit platform is unable to generate correct 64-bit PowerPC code

Operation	μ seconds
Attach	17.84 (0.16)
Component call	1.40 (0.02)
Detach	4.23 (0.49)

Table 1: **Interposer overhead.** There are three costs of interposition: attach, call, and detach. Attaching the interposer involves initializing the interposer and wrapper and updating the object translation table. Calls to the component involve two additional method calls to the wrapper object and a heap allocation. Detaching the interposer only involves updating the object translation table, making the cost to component callers zero (because they do not have any additional wait time); however, the process performing the detach must pay the given overhead for destroying the objects. The average cost of each operation is listed in microseconds along with its standard deviation.

accessed objects. The overhead of dispatching a virtual function call is approximately 10 cycles. The remainder of the overheads for online reconfiguration are from the specific implementations of interposition and hot-swapping. These overheads were measured on the 270 Workstation.

Interposition: There are three performance costs for interposition: attaching the wrapper, calling through the wrapper (as opposed to instrumenting the component directly), and detaching the wrapper. To measure these costs, we attached, called through, and detached an empty wrapper 100,000 times, calculating the average time for each of the three operations. Because the empty wrapper performs no operations (simply returning from the PRECALL and POSTCALL), all of the call overhead is due to interposition.

Table 1 lists the costs of interposition. Attaching the interposer is the most expensive operation, involving memory allocation and object initialization; however, at no point during the attach are incoming calls blocked. Although detaching the interposer only requires updating the object translation table, the teardown of the interposer and wrapper is listed as an overhead for the process performing the detach. One simple optimization to component calls is to skip the POSTCALL whenever possible. Doing so removes the expensive memory allocation, because no state would be kept across the forwarded call (control can be returned directly to the original caller).

Hot-swapping: K42's *file cache manager* objects (FCMs) track in-core pages for individual files in the system. To determine the expected performance of hot-swapping, we perform a "null" hot-swap of an FCM (swapping it with itself) at points of high system contention while running a 4-way SDET. Contention is detected by many threads accessing an FCM concurrently. Although high system contention is the worst time to swap (because threads are likely to block, increasing the

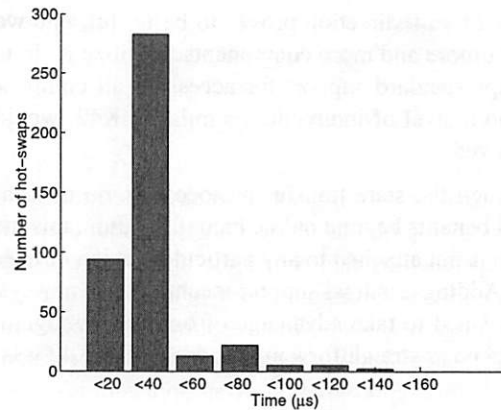


Figure 4: **Null swap.** This figure presents a histogram showing the cost of performing a null-swap of the FCM module at contended points of system execution. For each bin, there is a count of the number of swaps with completion times that fell within that bin. On average, a swap took 27.6 μ s to complete, and no swap took longer than 132.6 μ s.

duration of the mediation phases), it is important to understand this sort of "worst-case" swapping scenario.

During a single run of 4-way SDET the system detected 424 points of high contention. The average time to perform a hot-swap at these points was 27.6 μ s with a 19.8 μ s standard deviation, a 1.06 μ s minimum and a 132.6 μ s maximum. Hot-swapping while the system is not under contention is more efficient, because the forward and block phases are shorter. Performing random null hot-swaps throughout a 4-way SDET run gave an average hot-swap time of 10.8 μ s. Because most of the time spent doing a hot-swap is spent waiting for the generation to advance (during which no threads are blocked) the affect of these hot-swaps on the throughput of SDET is negligible.

6.3 Reconfiguration for performance

This section evaluates online reconfiguration's flexibility by using it to implement four well-known adaptive performance enhancements. Because these algorithms are well-known, the focus of this section is not on how the adaptive decisions were made, but rather the fact that online reconfiguration can be used to quickly and efficiently implement each of them.

Single v. Replicated FCMs: This experiment uses online reconfiguration to hot-swap between different component implementations for different workloads. We use two FCM implementations, a *single* FCM designed for uncontended use, and a *replicated* FCM designed to scale well with the number of processors. Although a single FCM uses less memory, it must pay a performance penalty for cross-processor accesses. A replicated FCM creates instances on each processor where it is accessed,

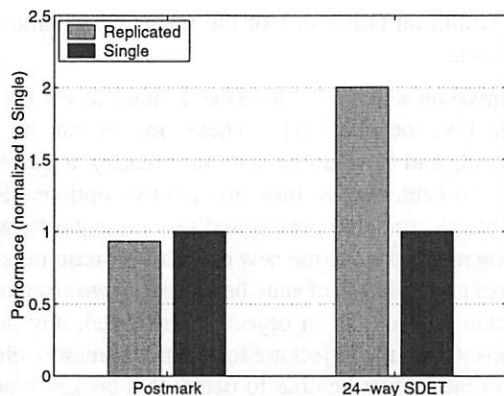


Figure 5: Single v. Replicated FCMs. This figure shows two different FCM implementations run under different workloads. In postmark, the single-access FCM performs better because it has less memory overhead during the creation and deletion of files. Conversely, the replicated FCM performs better under 24-way SDET because it scales well to multiple processors.

but at the cost of using additional memory.

Figure 5 shows the performance of each FCM under both Postmark (using 10,000 files, 50,000 transactions and file sizes ranging from 512B to 16KB) and 24-way SDET. Because Postmark is a single application that acts on a large number of temporary files, the overhead of doing additional memory allocations for each file with a replicated FCM causes a 7% drop in performance. On the other hand, using the replicated FCM in the concurrent SDET benchmark gives performance improvements that scale from 8% on a 4-way SDET to 101% on a 24-way SDET. A replicated FCM helps SDET because each of the scripts run on separate processors, but they share part of their working set.

K42 detects when multiple threads are accessing a single file and hot-swaps between FCM implementations when appropriate. Using this approach, K42 achieves the best performance under both workloads.

Exclusive v. Shared File Sessions: This experiment uses online reconfiguration to swap between an efficient non-shared component and a default shared component for correctness. We used a file handle implementation that resides entirely within the application. While this improves performance, it can only be used when an application has exclusive access to the file. Once file sharing begins, an in-server implementation must be swapped in to maintain the shared state.

Figure 6 shows the performance of swapping in the shared access version only when necessary in Postmark. Because most of the accesses in Postmark are exclusive, a 34% performance improvement is achieved.

Small v. Large Files: This experiment uses online reconfiguration to hot-swap between a specialized non-

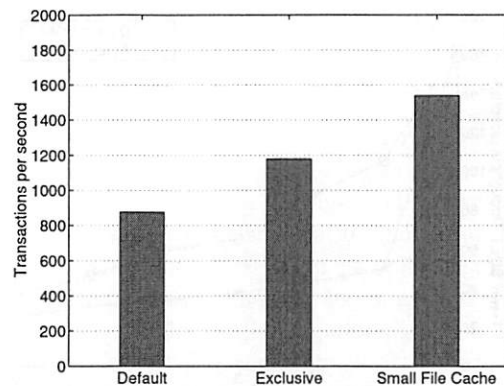


Figure 6: Common-case optimization. Using online reconfiguration, K42 can cache exclusive file handles within the application and swap to a shared implementation when necessary for correctness. A further enhancement is to cache small, exclusive files within the application's address space. This figure compares these three system configurations (default, exclusive, and small file cache) using Postmark. Using online reconfiguration for the exclusive case shows a 34% performance improvement, while the small file caching shows an additional 40% improvement beyond that.

shared component and a default shared component. In general, file data is cached with the operating system; however, access for small, exclusive files (< 3 KB) can be improved by also caching the file's data within an application's address space. While this incurs a memory overhead for double caching the file (once in the application, once in the OS), this is acceptable for small files, and it leads to improved performance.

Figure 6 shows the Postmark performance of three schemes: the default configuration, the exclusive caching scheme presented above, and application-side caching. We found that hot-swapping between caching implementations gives an additional 40% performance improvement over the exclusive access optimization.

Originally, K42 implemented this using a more traditional adaptive approach, hard-coding the decision process and both implementations into a single component. Anecdotally, we found that reimplementing this using online reconfiguration simplified and clarified the code, and it was less time-consuming to implement and debug.

Adaptive Page Replacement: This experiment uses online reconfiguration to implement an adaptive page replacement algorithm. An interposed wrapper object watches an FCM for sequential page mappings. If a sequence of more than 20 pages are requested, the access is deemed sequential, and the system hot-swaps to a sequentially optimized FCM that approximates MRU page replacement. If this sequential behavior ends (more than 10 non-sequential pages are requested), the FCM is swapped back to the default FCM.

Figure 7 shows the performance of 1-way SDET in the

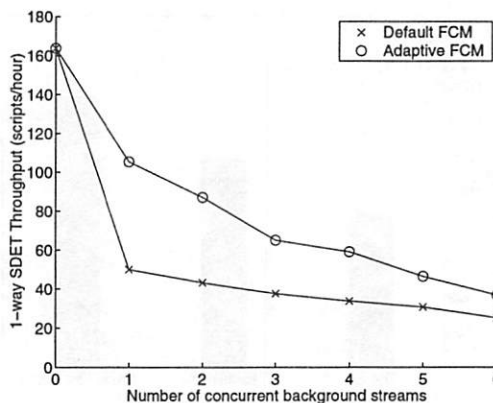


Figure 7: **Sequential page faults.** One adaptive page replacement algorithm performs MRU page replacement for sequential streams. This reduces the amount of memory wasted by streams whose pages will never be accessed again. Using online reconfiguration, K42 can detect sequential streams and swap to a sequentially optimized FCM. This figure compares the default page replacement to the adaptive algorithm by running 1-way SDET concurrently with a number of streaming applications. Using the adaptive page replacement, the system degrades more slowly, reducing the effect of streaming applications on the performance of the entire system.

face of competing streaming applications all run over NFS. These streaming applications access files significantly larger than the 512MB of main memory available in the 270 Workstation used in this experiment. Using the default FCM, the streaming applications quickly fill the page cache with useless pages, incurring the pager overhead immediately. This ruins SDET performance. On the other hand, using the adaptive approach, the sequential applications consume very little memory, because they throw away pages shortly after using them. This makes SDET performance degrade more slowly, meaning more streaming applications can be run while achieving the same performance as with the default FCM.

7 Related Work

Modifying the code of a running system is a powerful tool that has been explored in a variety of contexts. The simplest and most common example of adding new code to a running system is dynamic linking [24]. When a shared library is updated, all programs dependent on the library are automatically updated when they are next run. It is also possible to update the code in running systems using shared libraries; however, the application itself must provide this support and handle all aspects of the update beyond loading the code into memory.

Although several online reconfiguration methods exist in the middleware community [34], many do not transfer well to the realm of OSes due to their increased constraints on timing and resources. Identifying and implementing an online reconfiguration mechanism that works

well within an OS is one of the major contributions of this work.

Hjálmtýsson and Gray describe a mechanism for dynamic C++ objects [31]. These objects can be hot-swapped, and they do so without creating a quiescent state. To achieve this, they provide two options. First, old objects continue to exist and service requests, while all new requests go to the new object. This requires some form of coordination of state between the two co-existing objects. Second, if an object is destroyed, any active threads within the object are lost. For this reason, clients of an object must be able to detect this broken binding and retry their request.

CORBA [8], DCE [42], RMI [46], and COM [13] are all application architectures that support component replacement during program execution. However, these architectures leave the problems of quiescence and state transfer to the application, providing only the mechanism for updating client references.

Pu, et al. describe a “replugging mechanism” for incremental and optimistic specialization [43], but they assume there can be at most one thread executing in a swappable module at a time. In later work, that constraint is relaxed but is non-scalable [38].

In addition to the work done in different reconfiguration mechanisms, many groups have applied online reconfiguration to systems and achieved a variety of benefits. Many different adaptive techniques have been implemented to improve system performance [2, 25, 35]. Extensible operating systems have shown performance benefits for a number of interesting applications [7, 19, 49]. Technologies such as compiler-directed I/O prefetching [10] and storage latency estimation descriptors [39] improve application performance using detailed knowledge about the state of system structures. Incremental and optimistic specialization [43] can remove unnecessary logic for common-case accesses. K42’s online reconfiguration can simplify the implementation of these improvements, removing the complicated task of instrumenting the OS with the necessary hooks to do reconfiguration on a case-by-case basis.

K42 is not the first operating system to use an object oriented design. Object-oriented designs have helped with organization [27, 47], extensibility [11], reflection [54], persistence [15], and decentralization [1, 51]. In addition, K42’s method of detecting a quiescent state is not unique. Sequent’s NUMAQ used a similar mechanism for detecting quiescent state [37], and recently, SuSE Linux 7.3 has integrated a mechanism for detecting quiescence in kernel modules [36].

7.1 Open Issues

Although our prototype provides a solid base for numerous OS enhancements, there are a number of open issues that could expand the utility of K42's online reconfiguration. This section describes a number of these issues, and how they have been addressed in other systems.

Object creation and management: When a performance upgrade or security patch is released for a particular component implementation, every instance of that component should be hot-swapped. Additionally, any place where an instance of the component is created must also be updated to create the new component type rather than the old one. A common solution to this is an object "factory" that is responsible for creating and managing objects in the system. When an upgrade is requested, the factory is responsible for locating and upgrading each instance. Another similar solution is to use a garbage collector to track and update objects [23].

Coordinated swapping: While hot-swapping individual components can provide several benefits, there are times when two or more components must be swapped together. For example, an architecture reconfiguration that updates the interfaces between two objects must swap the objects concurrently. Several different groups have looked into how to perform this while ensuring correctness [9, 14, 30].

Confirming component functionality: Although K42 requires that swapped components support the same interface, it makes no guarantees that the functionality provided by the two components is the same. Although it may be possible for components to provide annotations about their functionality, or show type-safety [23], component validity has been proven to be generally undecidable [29]. Because of this, systems must make a tradeoff between flexibility and provable component correctness.

The most common method for improving component correctness is through type safety. Unfortunately, this generally reduces the flexibility or performance of potential reconfigurations. Dynamic ML [23] provides type safety guarantees for components with unchanging external interfaces, but, its reliance upon a two-space copying garbage collector makes it unreasonable for use in an operating system environment. Hicks [30] provides type safety guarantees for components with external data and changing interfaces, but relies upon programmer defined "safe" swap-points and requires that every instance of a component be swapped (thus two versions of a component cannot exist within the system at once).

Interface management: Currently, K42's online reconfiguration requires that swapped components support the same interface. While it is straightforward to expand

these interfaces (because the old interface is a subset of the new interface), it is currently not possible to reduce interfaces in K42; in particular, it is not possible to know if an active code path in the system relies upon a particular part of the interface. Several systems have looked into allowing interface changes using type-safety and programmer defined safe-points for updates to provide the necessary information about component usage [16, 23, 30].

Generic state transfer: K42's hot-swapping mechanism provides a protocol for negotiating the best common format for state transfer between objects. But, it relies upon support from the components being swapped to complete state transfer. Because this becomes increasingly complex as the number of implementations increases, it would be ideal if the infrastructure could perform the entire state transfer, making hot-swapping entirely transparent. While this goal may not be fully attainable, it may be possible to provide more support than K42 currently does. For example, Hicks examines the possibility of automatically generating state transformation functions for simple cases [30].

Avoiding quiescence: It may be possible to instantiate the new object and have it start processing calls while the old object completes calls that are still in-flight. In some cases this would require a way to maintain coherence between the states of the two objects; however, in other cases it may be possible to do a lazy update of the new object's state after in-flight calls have completed.

8 Conclusions

Online reconfiguration provides an underlying mechanism for component extension and replacement through interposition and hot-swapping. These mechanisms can be leveraged to provide a variety of dynamic OS enhancements. This paper identifies four support mechanisms used for interposition and hot-swapping, and describes their implementation in the K42 operating system. We demonstrate the flexibility of online reconfiguration by implementing an adaptive paging algorithm, two common-case optimizations, and a workload specific specialization.

Acknowledgements

A large community has contributed to the K42 project over the years, and we would like to thank everyone that helped us get the system to the state where the research presented in this paper was possible. Craig Soules was supported by a USENIX Fellowship. Jonathan Appavoo was supported by an IBM Fellowship.

References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. Summer USENIX Technical Conference, July 1986.
- [2] J. Aman, C. K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger. Adaptive algorithms for managing a distributed data-processing workload. *IBM Systems Journal*, 36(2):242–283, 1997.
- [3] J. Appavoo, M. Auslander, D. Edelson, D. D. Silva, O. Krieger, M. Ostrowski, B. Rosenberg, R. W. Wisniewski, and J. Xenidis. Providing a Linux API on the scalable K42 Kernel. Page to appear.
- [4] J. Appavoo, K. Hui, M. Stumm, R. W. Wisniewski, D. D. Silva, O. Krieger, and C. A. N. Soules. An infrastructure for multiprocessor run-time adaptation. ACM SIGSOFT Workshop on Self-Healing Systems. ACM, 2002.
- [5] M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm. Customization lite. Hot Topics in Operating Systems, pages 43–48. IEEE, 1997.
- [6] J. K. Bennett. *Distributed Smalltalk: inheritance and reactivity in distributed systems*. PhD thesis, published as 87–12–04. Department of Computer Science, University of Washington, December 1987.
- [7] B. N. Bershad, S. Savage, P. Pardy, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, 29(5), 3–6 December 1995.
- [8] C. Bidan, V. Issarny, T. Saridakis, and A. Zaras. A dynamic reconfiguration service for CORBA. International Conference on Configurable Distributed Systems, pages 35–42. IEEE Computer Society Press, 1998.
- [9] T. Bloom. *Dynamic module replacement in a distributed programming system*. PhD thesis, published as MIT/LCS/TR–303. Massachusetts Institute of Technology, Cambridge, MA, March 1983.
- [10] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2):111–170. ACM, 2001.
- [11] R. H. Campbell and S.-M. Tan. μ Choices: an object-oriented multimedia operating system. Hot Topics in Operating Systems, pages 90–94. IEEE Computer Society, 4–5 May 1995.
- [12] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. *Modula-3 report (revised)*. 52. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 1 November 1989.
- [13] Distributed Component Object Model Protocol-DCOM/1.0, <http://www.microsoft.com/Com/resources/comdocs.asp>.
- [14] R. P. Cook and I. Lee. DYMO: A dynamic modification system. Pages 201–202.
- [15] P. Dasgupta, R. J. LeBlanc, Jr, M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 24(11):34–44, November 1991.
- [16] D. Duggan. *Type-based hot swapping of running modules*. Technical report SIT-CS–2001–7. October 2001.
- [17] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. ACM Symposium on Operating System Principles. ACM, 2001.
- [18] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: application-level virtual memory. Hot Topics in Operating Systems, pages 72–77. IEEE Computer Society, 4–5 May 1995.
- [19] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an operating system architecture for application-level resource management. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, 29(5), 3–6 December 1995.
- [20] M. E. Fiuczynski and B. N. Bershad. An extensible protocol architecture for application-specific networking. USENIX. 1996 Annual Technical Conference, pages 55–64. USENIX. Assoc., 1996.
- [21] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. Symposium on Operating Systems Design and Implementation, pages 87–100, February 1999.
- [22] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems*, 20(1):49–83. ACM, February 2002.
- [23] S. Gilmore, D. Kirli, and C. Walton. *Dynamic ML without dynamic types*. Technical report ECS-LFCS–97–378. June 1998.
- [24] R. A. Gingell. Shared libraries. *UNIX Review*, 7(8):56–66, August 1989.
- [25] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1997.
- [26] J. Gosling and H. McGilton. *The Java language environment*. Technical report. October 1995.
- [27] A. S. Grimshaw, W. Wulf, and T. L. Team. The Legion vision of a world-wide virtual computer. *Communications of the ACM*, 40(1):39–45. ACM Press, January 1997.
- [28] Guardian Digital, Inc., <http://www.guardiandigital.com/>.
- [29] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131. IEEE, February 1996.
- [30] M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. ACM, 2001.
- [31] G. Hjalmysson and R. Gray. Dynamic C++ classes: a lightweight mechanism to update code in a running program. Annual USENIX Technical Conference, pages 65–76. USENIX Association, 1998.
- [32] The K42 Operating System, <http://www.research.ibm.com/K42/>.
- [33] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.
- [34] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38. ACM Press.
- [35] Y. Li, S.-M. Tan, Z. Chen, and R. H. Campbell. *Disk scheduling with dynamic request priorities*. Technical report. University of Illinois at Urbana-Champaign, IL, August 1995.
- [36] P. E. McKenney, J. Appavoo, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read copy update. Ottawa Linux Symposium, 2001.
- [37] P. E. McKenney and J. D. Slingwine. Read-copy update: using execution history to solve concurrency problems. International Conference on Parallel and Distributed Computing and Systems, 1998.
- [38] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems (TOCS)*, 19(2):217–251. ACM Press.
- [39] R. V. Meter and M. Gao. Latency management in storage systems. Symposium on Operating Systems Design and Implementation, pages 103–117. USENIX Association, 2000.
- [40] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of techniques to improve protocol processing latency. ACM SIGCOMM Conference. Published as *Computer Communication Review*, 26(4):73–84. ACM, 1996.
- [41] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. Symposium on Operating Systems Design and Implementation, pages 229–243. Usenix Association, Berkeley, CA, October 1996.
- [42] *Distributed Computing Environment: overview*. OSF-DCE-PD-590-1. Open Software Foundation, May 1990.
- [43] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: streamlining a commercial operating system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, 29(5), 3–6 December 1995.
- [44] Read-Copy Update Mutual Exclusion for Linux, <http://lse.sourceforge.net/locking/rupdate.html>.
- [45] Posting by Linus Torvalds to linux-kernel mailing list: Summary of changes from v2.5.42 to v2.5.43, <http://marc.theaimsgroup.com/?l=linux-kernel&m=103474006226829&w=2>.
- [46] Java Remote Method Invocation - Distributed Computing for Java, November 1999. <http://java.sun.com/marketing/collateral/javarmi.html>.
- [47] V. F. Russo. *An object-oriented operating system*. PhD thesis, published as UIUCDCS–R–91–1640. Department of Computer Science, University of Illinois at Urbana-Champaign, January 1991.
- [48] Security-Enhanced Linux, <http://www.nsa.gov/selinux/index.html>.
- [49] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. *An introduction to the architecture of the VINO kernel*. Technical report. 1994.
- [50] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. Symposium on Operating Systems Design and Implementation, pages 213–227. Usenix Association, Berkeley, CA, 28–31 October 1996.
- [51] J. A. Stankovic and K. Ramamritham. The Spring kernel: a new paradigm for real-time operating systems. *Operating Systems Review*, 23(3):54–71, July 1989.
- [52] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, 27(5):203–216. ACM, 1993.
- [53] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: application-specific handlers for high-performance messaging. ACM SIGCOMM Conference, August 1996.
- [54] Y. Yokote. The Apertos reflective operating system: the concept and its implementation. Object-Oriented Programming: Systems, Languages, and Applications, pages 414–434, 1992.

Checkpoints of GUI-based Applications

Victor C. Zandy and Barton P. Miller
Computer Sciences Department
University of Wisconsin – Madison, USA
{zandy,bart}@cs.wisc.edu

Abstract

We describe a new system, called *guievict*, that enables the graphical user interface (GUI) of any application to be transparently migrated to or replicated on another display without premeditative steps such as re-linking the application program binary or re-directing the application process's window system communication through a proxy. *Guievict* is based on a small X window server extension that enables an application to retrieve its *window session*, a transportable representation of its GUI, from the window server and a library of GUI migration functionality that is injected in the application process at run time. We discuss the underlying technical issues: controlling and synchronizing the communication between the application and the window system, identifying and retrieving the GUI resources that form the window session, regenerating the window session in a new window system, and maintaining application transparency. We have implemented *guievict* for the XFree86 implementation of the X window system. The GUI migration performance of *guievict* is measurably but not perceptibly worse than that of a proxy-based system.

1 INTRODUCTION

Application mobility is the ability for an application in execution to follow its user, whether the user moves from one computer to another or moves with their computer around the Internet. This mobility includes moving the executing code, network connections, access to the graphical user interface (GUI) or the GUI itself, and access to files. We are developing a *roaming applications* system, called *evict*, that addresses these issues, providing application mobility that is transparent to users and applications while requiring no modification to system code.

This paper describes *guievict*, the part of the *evict* system that migrates an application's GUI. An applica-

tion's GUI might migrate as the application itself migrates, or independently of the application. Both types of migration require the ability to capture the application's *window session*, a transportable representation of its GUI. Window sessions are difficult to capture because not all of the state they represent is resident in the application. Portions of an application's GUI state reside in the window server, a separate system that handles the display of all GUI-based applications on the same host. Most window servers do not provide a way to extract the state of an individual application's resources.

Guievict has been implemented for the X window system [19] and has the following characteristics:

- ❑ Migration occurs at application granularity; users can select and move the GUI of individual applications from their desktop, leaving other application GUIs behind or free to move elsewhere.
- ❑ Any application program, including those based on legacy toolkits, can be migrated without modifications such as re-programming, re-compiling, or re-linking.
- ❑ Migration can be unpremeditated; users do not need to run their applications in a special way, such as by redirecting their GUI communication through a proxy.
- ❑ No modifications to window system code are required; our functionality is contained in the window server extension, which does not require the server to be re-linked, and a library that is loaded in the application at run-time.
- ❑ The *guievict* functionality can also be used to replicate the GUI of individual applications on demand on multiple desktop hosts, enabling multiple users to interact with a single instance of an application for collaborative work or to support remote service.

There are several essential elements to a GUI-based application (see Figure 1). The application runs in one or more processes on the *execution host* and the user interacts with it from a possibly different *desktop host*. The desktop comprises a display, keyboard, and mouse managed by a window system that multiplexes the desk-

This work is supported in part by Department of Energy Grants DE-FG02-93ER25176 and DE-FG02-01ER25510, Lawrence Livermore National Lab grant B504964, NSF grants CDA-9623632 and EIA-9870684, and VERITAS Software. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

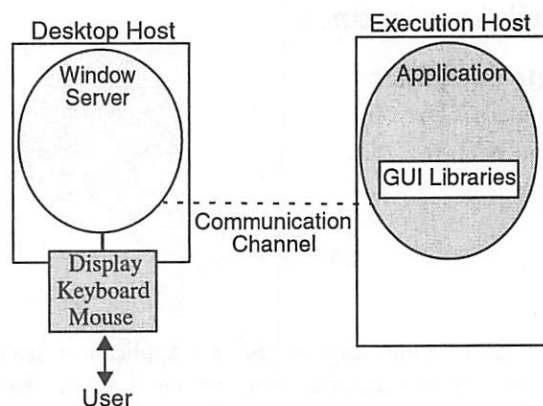


Figure 1: The elements of a GUI-based application.

The user interacts with the application through hardware managed by the window server on the desktop host. The application process executes on a possibly different execution host, exchanging GUI-related messages with the window server over a communication channel.

top for all applications that interact with the user at that host. The window system responds to requests for GUI services (such as creating a window) sent by the application and passes notification of desktop events (such as a mouse click) to the application over a communication channel such as a network connection or (when the execution and desktop hosts are the same) shared memory. The state of the application GUI is distributed between the window system and a library (often called a toolkit) in the application process.

Our focus is on the migration of a single application from one desktop host to another. Our abstraction encapsulates precisely the GUI state of the *application* — not individual windows of the application, nor the windows of all applications that are being served for the user on the desktop host. Many previous systems have studied techniques for *session migration*, in which a user's entire desktop is migrated to another machine [7,14,16,17,23]. Our ability to migrate at application granularity has several advantages over session migration. First, it is flexible: it gives users the freedom to migrate only the applications they need at their new location, saving migration costs, and it enables users to work with multiple desktops at the same time. Second, session migration systems generally involve a virtualization layer, such as a virtual machine [7,14] or nested window server [16,17], a sort of GUI prison to which the user must redirect their applications in advance. In addition to the inherent overhead of virtualization, these layers often emulate basic and generic display hardware, preventing applications from taking advantage of enhancements or acceleration features present in the real underlying hardware. Third, a mechanism for applica-

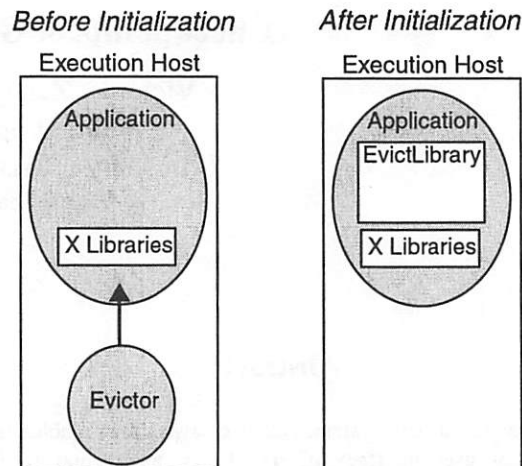


Figure 2: Initializing the application process.

The evictor stops the application process, forces it to load the evict client library, and exits.

tion migration enables other useful operations, such as a *GUI attach* that allows other users to dynamically attach to and interact with the GUI of another user's application.

Guievict offers two major advantages over previous systems that perform GUI migration at application granularity, most notably xmove [22]. First, it enables users to migrate their applications without the premeditative step of redirecting the application to the xmove proxy. Although the details of the redirection can be hidden from the user in a launch script, the user still must use the script to start applications that they anticipate moving. In contrast, guievict offers application mobility features without asking users to develop new habits and foresight. Second, it simplifies process migration of GUI-based applications. Although migration was not one of its original design goals, xmove can be combined with process checkpointing to migrate a GUI-based application. However, in addition to the application process, the xmove proxy and the state of the communication channel between the proxy and the application must be checkpointed and migrated. With guievict, the window session and mechanisms that manage its migration reside in the address space of the application process, where it can be migrated along with application code and data.

To migrate the GUIs of ordinary, unmodified applications from one display host to another, we have overcome four main challenges:

- ❑ Dynamically taking control of the window system communication of a running program: We inject code into the application process that discovers the process's communication channel to the window server and synchronizes its communication with the server.

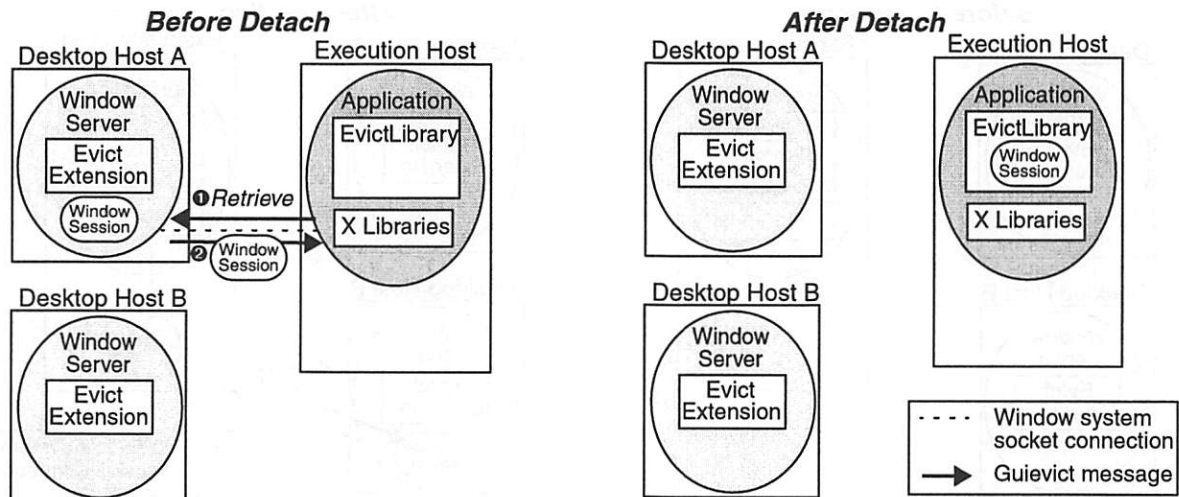


Figure 3: Detaching a GUI from Desktop A.

The evict client library requests the window session from the evict server extension at Desktop A, then closes connection.

- ❑ Retrieving the GUI resources of an application: We have developed an extension to the X window server that enables an application, at any time, to determine the identifiers of its GUI resources and the dependencies among these resources, and to extract these resources in a form from which they can be regenerated on a new desktop.
- ❑ Regenerating an application's resources in another desktop host: We use standard X protocol operations to regenerate GUI resources in the new window server.
- ❑ Ensuring GUI migration is transparent to the application: Applications whose GUIs have been migrated can be confused by the resulting changes to resource identifiers, message sequence numbers, and display characteristics such as pixel depth. We interpose a filter (called the *guimux*) on the communication between the application that provides the mapping necessary to maintain transparency. The *guimux* also serves as a multiplexor for GUI replication.

The main limitations of *guievict* are that (1) it requires the user to install our X window server extension on their desktop hosts, (2) it requires the availability of symbols for the window protocol stubs used by the application, and (3) it has a large (20 second) overhead in checkpointing font state. We discuss implications and possible workarounds of the last two issues in Section 3.

The remainder of this paper is organized as follows. Section 2 presents the architecture of *guievict*. Section 3 describes its implementation. Section 4 presents our evaluation of *guievict*. Section 5 identifies the security

issues raised by *guievict*. Section 6 presents related work.

2 SYSTEM OVERVIEW

The major steps in the operation of *guievict* are: initializing the system, migrating an application's GUI from one desktop host to another, replicating an application's GUI on multiple desktops, and migrating an application process along with its GUI. These operations involve four system components. The *evictor* is a program that the user runs to control the evict system. The user runs it on the same host as the application process. The *evict client library* is loaded by the evictor at run time into the application process and implements application-side GUI migration operations. The *evict server extension* is the corresponding server-side component. The *guimux* is a daemon, started by the client library, that runs on the execution host and ensures that GUI migration and replication is transparent to the application.

In the remainder of this section we describe the *guievict* operations in detail. In Section 3, we discuss our solutions to the technical problems underlying these operations.

2.1 Initialization

The user prepares a running application process for the evict system by invoking the evictor's *initialize* operation, passing the application process id as an argument. The evictor *hijacks* [25] the application process: it stops the process and forces it to load and initialize the evict client library (see Figure 2). During its initialization the client library establishes a communication channel for future interaction with the user. Hijacking is transparent

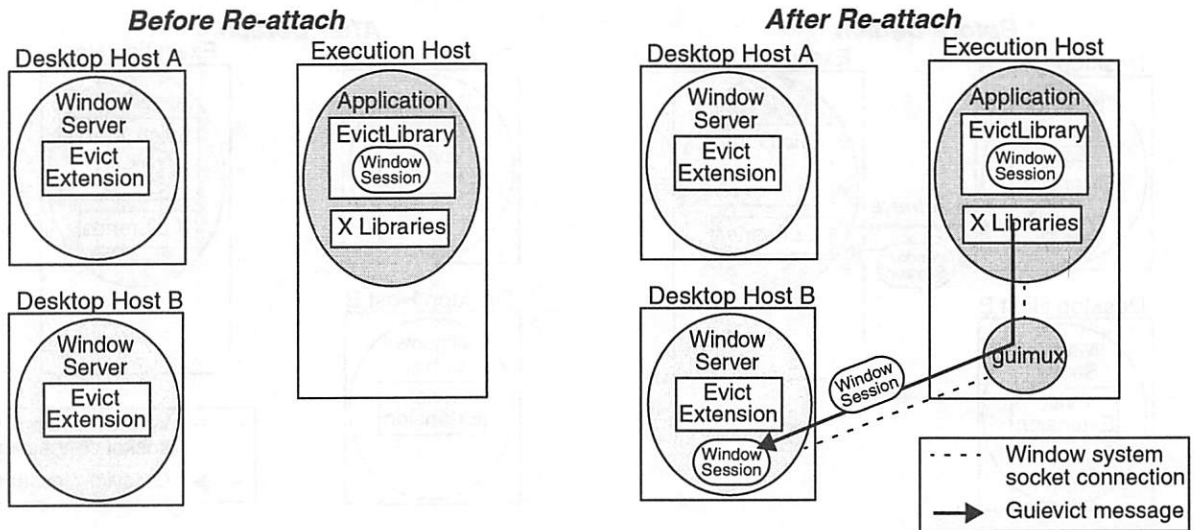


Figure 4: Re-attaching a GUI to Desktop B.

The evict client library establishes a new connection to Desktop B through the guimux, which forwards all window communication between the application and Desktop B, starting with the request to regenerate the window session.

to the application; afterward the evictor resumes the application process, allowing it to continue normally. At any later time the user can run the evictor again to request an evict operation. The evictor interrupts the application process and sends commands to the client library over the communication channel.

The evict server extension must be loaded and initialized in the window server before the user's first request for an evict operation. The XFree86 window server does not support run-time extension loading, so in our implementation the server must be configured to load the evict extension before it is started, but this is not necessary in general.

2.2 GUI Migration

GUI migration is broken down into two steps.

First, the user requests the application (with an evictor command) to *detach* its GUI from the window server. In response the evict client library (see Figure 3):

- Synchronizes the application's communication with the window server and blocks the application from further communication;
- Retrieves the window session from the server;
- Closes the connection to the window server.

Second, the user requests the application to *re-attach* its GUI to a new window server (see Figure 4). The evict client library starts a guimux process, replaces the application's socket to the window server with a full-duplex pipe to the guimux process, and then transfers control to the guimux process. Then the guimux process:

- Opens a connection to the new window server;
- Regenerates the state of the window session;
- Signals the evict client library to resume the application.

The re-attach operation may come an arbitrary period of time after the detach. In the meantime, the client library suspends the execution of application code to prevent the temporary absence of a window server connection from affecting the application. For the common case of a user who wishes to detach from one desktop and attach to another in one logical operation, the evictor provides a combined detach and attach command.

2.3 GUI Replication

GUI replication is a simple variation of GUI migration. The user requests (with the evictor) the application to *replicate* its GUI on another window server. The evict client library performs all but the final step of the detach operation. That is, it acquires the current state of the window session, but does not close the connection to the window server. It then performs a normal attach operation to connect the GUI to the additional window server (see Figure 5). If this is the first time an attach operation has been performed, it also redirects the original window server connection through the guimux process.

2.4 GUI+Process Migration

Guievict supports the simultaneous migration of the application process and its GUI. Figure 6 shows a typical scenario for this type of mobility in which the execution and desktop hosts are the same (such as a laptop) and the user wants to migrate their application process

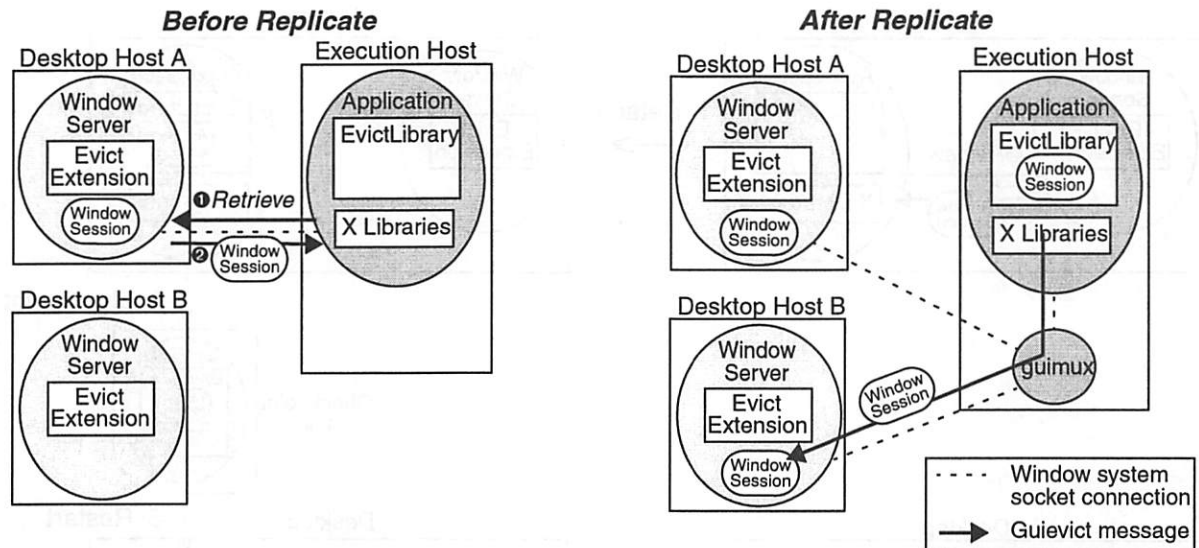


Figure 5: Replicating a GUI on Desktop B.

Replication is similar to migration, except that the connection to Desktop A is preserved and multiplexed by guimux with the connections to other desktop hosts.

and its GUI to another host (such as the computer on their desk). In this scenario, the user uses evictor to request the application to *migrate*, providing two arguments: the name of the new X window server and the name of the new execution host. Evict then:

- Detaches the application's GUI from its window server;
- Terminates the guimux daemon (if one is running);
- Checkpoints the application process, producing a *checkpoint file* [15] containing the state of the application process, including its window session;
- Exits the application process.

At this point the user must transport the checkpoint file to the new execution host and invoke the evictor to *restart* the application process. To complete the migration, evict:

- Restores all the state of the application process except for its GUI;
- Attaches the application process to the new window server.

Checkpointing and restarting the application process is transparently performed by a user-level process checkpoint library that is linked with the evict client library. The details of this library have been described previously [15,25] and are outside the scope of this paper.

3 IMPLEMENTATION

We have implemented evict on x86 Linux using the XFree86 implementation of the X window system. We

describe the major technical issues and how we solve them in our implementation: hijacking the application, find the application's connection to the window server, synchronizing its connection, retrieving and restoring GUI resources, and ensuring that GUI migration is transparent to the application.

3.1 Hijacking the Application

Process hijacking can be safely implemented with basic dynamic instrumentation mechanisms, such as those provided by the Dyninst API [6]. These include stopping the process, forcing the process to execute code to load and initialize the evict client library, and then resuming the process. The evictor contains its own implementation of these mechanisms to avoid requiring users to install additional software like the Dyninst API (which contains much more functionality than evict requires).

The evictor forces the application process to load the evict client library by injecting code that calls the run-time library loading feature (usually named *dlopen*) of the process's dynamic loader. This technique does not directly work with statically-linked programs, since there is no dynamic loader in processes based on such programs. Supporting statically-linked programs is important because many GUI-based applications are distributed statically to avoid forcing users to have the necessary GUI library dependencies. We have developed hijacking functionality to cope with statically linked programs. The idea is to map and initialize a copy of the dynamic loader into the process's address space, essentially by reproducing the initial steps the

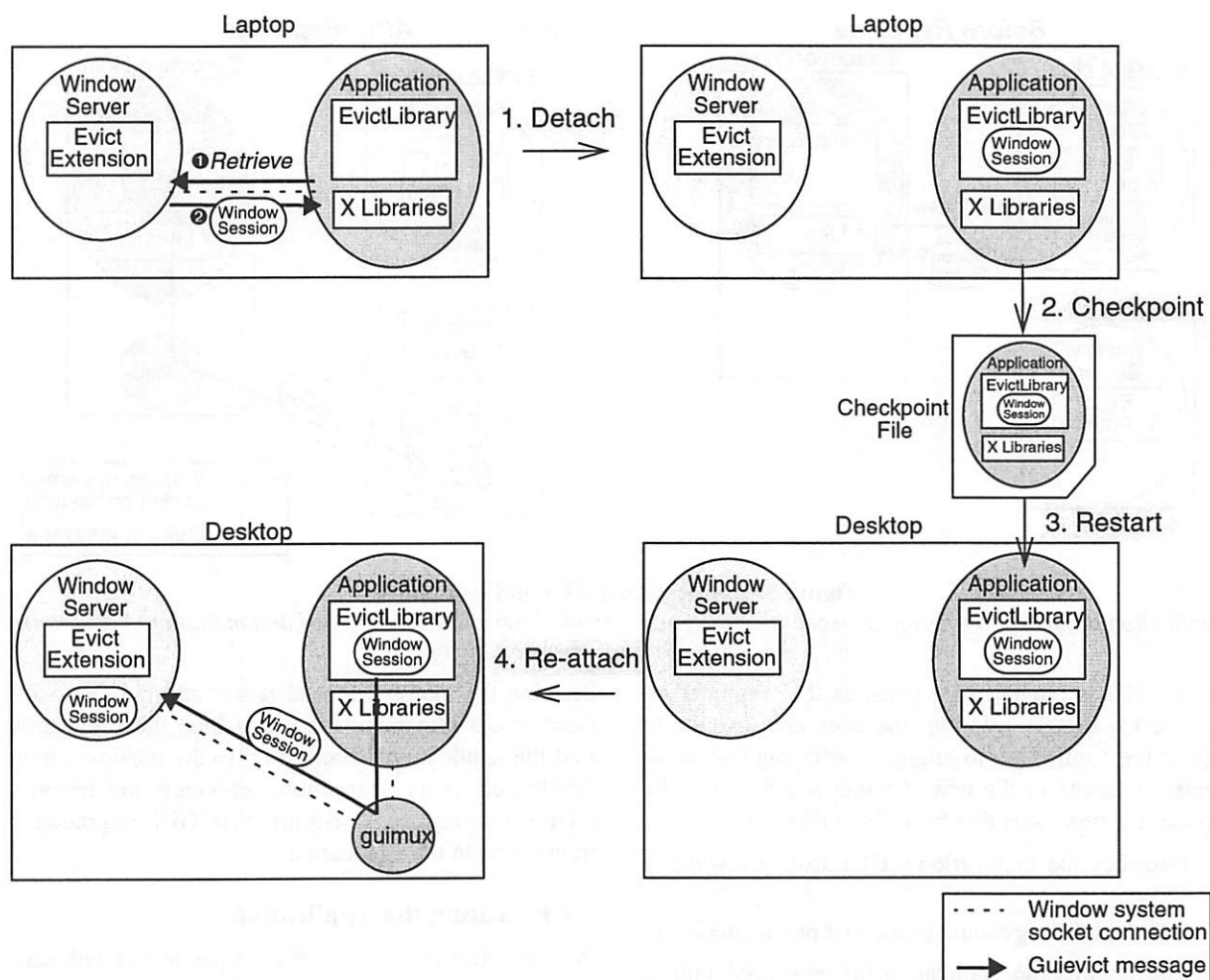


Figure 6: Migrating an application process and its GUI from a laptop to a desktop computer.

(1) The evict client library detaches the GUI from the window server; (2) Evict checkpoints the application process, producing a checkpoint file containing the entire application state including the window session; (3) After transporting the checkpoint file to the desktop host, evict restarts the application process; (4) Evict re-attaches the GUI to the desktop window server.

operating system takes when loading a dynamic-linked program. This instance of the dynamic loader does not control the original code in the process, but rather serves only to provide an implementation of dlopen that we can call as we do for dynamically-linked programs.

The evict client library creates a named Unix domain socket for subsequent communication with the evictor. The library deletes the socket when the application exits normally. The socket name is based on the application's process and user ids to avoid conflicts with other independently running instances of evict and to allow stale sockets left behind by abnormal termination to be cleaned up by the user who discovers them. The evictor gets the attention of the evict client library by writing a message to the socket, which causes the applica-

tion process to receive a signal that is handled by the library.

3.2 Finding the Window Server

X windows applications communicate with the window server over a Unix domain or TCP socket. Unlike proxy-based systems such as xmove, guievict may be invoked after the creation of the connection to the window server. It must search the file descriptors of the process for sockets connected to a window server. Most operating systems provide a way, such as a /proc entry on Linux, to list the open file descriptors of a process; on those that do not, guievict can test each possible file descriptor with the fstat system call. Guievict looks for a file descriptor that (1) refers to a socket inode (as

reported by the `fstat` system call), and (2) is connected to a window server.

The second condition is difficult to check. In many cases, the `getpeername` system call, which returns the address of the socket on the other side of a connection, is sufficient: we check that the socket peer address is one of the well-known X window server TCP ports or Unix domain socket names. This test can fail if the application is tunneled to the window server through a proxy, since the peer address of its socket will be the proxy's address. In many common proxy configuration, such as ssh tunnels [24] and firewall port forwarding rules [26], the difference between the proxy address and a normal window server address is a small positive offset in the port number, which is easy to recognize.

In the unlikely event that `getpeername` does not reveal an obvious window server connection, `guievict` checks whether the peer address of each socket leads to a window server. It creates a new socket, attempts to connect it to the peer address, and, if the connection succeeds, performs the first round of the standard X windows handshake. If the server gives the expected response to the handshake, `guievict` concludes that it has found a socket connected to an X server. If the probe fails on all sockets, `guievict` gives up control of the application.

The use of this probe raises two concerns. First, the probe may succeed on non X window servers that happen to respond to the handshake like a X windows server. In practice, the response is distinguishable from that seen in common protocols such as ssh, telnet, ftp, and http, however unfortunately it is not so distinctive to presume that conflicts will not occur in less common protocols. In the event of a false positive, `guievict` should eventually receive nonsensical messages from the server, after which it will abort. Second, the probe may have a negative effect on probed servers. Although server implementations should be robust to spurious connections, not all existing ones are, particularly those that have not been hardened for Internet deployment. For users who cannot risk using the probe, the evictor accepts a command line argument to identify the file descriptor or peer address of the window system connection.

3.3 Synchronizing Communication

`Guievict` must ensure that the state of the window session does not change while it is being retrieved. Changes to window state are caused by messages exchanged between the application and the window server. `Guievict` synchronizes the communication by finding a point in the message stream where there are no

partially sent or unanswered requests, and then blocks further communication.

The synchronization occurs in two steps. First, `guievict` forces the application process to reach a message boundary in the stream of messages from the application to the server. It examines the application's process stack before starting an operation, searching for X library functions that are stubs for X protocol requests. If such a function appears on the stack, which indicates that the application process may be in the middle of sending a message, `guievict` sets a timer, resumes the application code for a short period of time, and then re-examines the stack. This procedure repeats until the stack is free of potentially unsafe functions.

Second, `guievict` sends an X protocol request containing an illegal resource identifier to the server. The only effect of this request is that it elicits an error message from the server. `Guievict` reads and scans the stream of messages from the server until it recognizes the error, at which point the client has no unanswered requests and the connection is synchronized. The messages read before the error are buffered and re-sent to the application from the `guimux` when the application is allowed to resume.

Detecting the presence of X protocol stubs on the application process stack depends on the presence of the symbols for these functions in the application process. This is not an issue for dynamically linked applications because the symbols must be present to facilitate linkage. However, symbols may be stripped from statically linked executables. The evictor refuses to work with stripped static applications. To remedy this limitation we are investigating alternative approaches to synchronizing the communication that do not depend on stack traces, including inferring message boundaries by analyzing messages as they are sent over the socket.

3.4 Retrieving and Regenerating GUI Resources

X windows applications create, modify, and destroy GUI resources through the exchange of X protocol messages with the X window server. GUI resources reside in the window server and are indirectly manipulated by the application by 32-bit *resource identifiers*, which are drawn from a namespace that is global across all clients of a window server. Clients choose the low-order bits of the identifier for each resource that they create, but they must set the high-order bits to a fixed *client id* chosen by the server when the application connects to it.

It is generally impossible to locate the resource identifiers in the application process's code and data, so we must get them from the X window server. The window server manages a per-client table of allocated resources but, unfortunately, it does not provide external

access to the information in this table. Since no previously reported server extension has addressed this limitation, our server extension provides the missing interface. Using our *GetResources* request, an application can request the server to return an enumeration of all the resources that the application has created. For most types of resources, the resource identifier is sufficient for the application to retrieve the state of the corresponding resource with standard X protocol requests, but there are four exceptions: windows, graphics contexts, cursors, and fonts. The *GetResources* reply for windows includes the background pixel value and the window's cursor identifier. The replies for graphics contexts and cursors include their entire state: for a graphics context, a small array of flags, and for a cursor, its bit-map and geometry.

Fonts are more complicated. X windows fonts are stored at the server. Clients acquire the use of a font by sending a request to the server containing a font resource identifier and the name of the font. The server loads and binds the font to the identifier if it has the font, and otherwise returns an error. Applications can request detailed geometric information about the font associated with a font identifier, but unfortunately there is no request to map a font identifier to the name of the font. Strangely, the server discards the font name after loading a font, so the mapping is not possible even within our server extension. During the detach operation, the evict client library performs a search to map each font identifier to a font name. It requests the server to list of all of its stored fonts (a standard X protocol request), and then searches this list for a font whose geometry matches that of each font identifier. Usually, the font name suffices to regenerate the font resource on a new window server, but sometimes the new server does not have a font with that name. In those cases, guievict searches the font list on the new server and selects the font with the closest matching geometry, using a least-squares font matching algorithm similar to that used by HP Shared X [9]. This complicated and expensive approach to migrating font resources could be eliminated by switching to client-managed fonts, a recently proposed architectural change to the X window system [10]. In the meantime, we eliminate the overhead by caching in the application's file system the font names and geometry of each server we use regularly.

Sometimes it is not possible to regenerate pixel-based resources identically to their previous instances. Displays can vary by the number of bits per pixel (depth) and the method by which pixels are mapped to colors (visual type), both of which cause the meaning of pixel values to change. Xmove provided an additional translation operation that mapped pixel values from

their previous depth and visual to that available on the current server. Recent developments for the X server, however, promise to eliminate the need for such translation. In particular, the R&R extension and shadow framebuffers [11] are server-based mechanisms for virtualizing depth and visual that have been designed with the goal of providing heterogeneity support for migration and replication systems. Our extension complements these developments; together they combine to produce a system for GUI migration that is transparent to display heterogeneity.

3.5 Maintaining Transparency

The main role of the guimux daemon is to make GUI migration transparent to the application process by translating resource identifiers and sequence numbers that appear in the messages exchanged between the application and the window server. The resource identifier mapping is initialized during the regeneration of GUI resources. The evict client library regenerates a resource by issuing an ordinary X protocol resource creation request containing the original identifier. As guimux forwards these requests, it replaces the identifier with an unused identifier that is valid in the current window server. For subsequent messages from the application to the server, the guimux maps references to resources to their current identifier; it performs a similar reverse mapping on messages from the server to the application. As the application destroys resources, they are removed from the map.

Sequence numbers occur in messages sent from the window server to the application and represent the number of messages the server has processed for the application; they do not occur in messages from the application. The guimux replaces the sequence number of a message with the next sequence number expected by the application process. This procedure is initialized when guievict synchronizes the communication to the window server. At the point, the next sequence number expected by the application process is the sequence number contained in the sentinel error reply.

Replication of windows on multiple displays extends the role of the guimux process. While managing a replicated GUI, the guimux maintains a separate translation map for each window server connection. Messages from the application are translated and sent to each window server. Messages from the window servers are reverse translated and forwarded in series to the application. To control the behavior of replicated GUIs, guimux accepts a set of commands, sent by the evictor, that act as primitives for setting replication policy. For example, the user can suppress the forwarding of keyboard, mouse, and window modification events from

selected desktop hosts to allow users seated at those desktops to observe but not modify the state of the GUI; more sophisticated policies for managing collaborative work [2,12] can be built over these primitives.

4 EVALUATION

We have evaluated the performance of guievict's GUI migration functionality. As a point of reference, we compared its performance to the proxy-based xmove system. We measured the time to detach and re-attach a GUI and the impact on interactive response. We performed our measurements on a 700 MHz Pentium-III laptop running XFree86 4.2.0 on Linux 2.4.18, and we used the most recent release of xmove [21]. Overall, the results are not surprising. Guievict takes somewhat longer than xmove to detach a GUI from a window server, but re-attaches in comparable time. Xmove and guievict (after re-attach) both increase the latency of the communication between the application process and the window server, but not enough to be perceptible to users.

4.1 Detach and Re-attach Latency

We measured the latency of detaching a GUI from a window server and then re-attaching it to the same window server for several applications. The guievict detach latency is the elapsed time from when the evict client library receives the detach command to just after it closes the connection to the window server. The guievict re-attach latency is the elapsed time from when the evict library receives the re-attach command to just after it allows the application process to continue. The xmove latencies are analogous. We ran both the application process and the window server on the laptop and we detached the application's GUI after its initial windows were drawn but before any user interaction with the GUI. We report average measurements over five runs. Our results are reported in Table 1.

Guievict has a more expensive detach operation than xmove. Table 2 breaks down the average guimux detach time for one application. The most expensive stage is mapping the font identifiers to fonts names, during which most of the time is spent waiting for the server to return the complete list of the fonts. More generally, guievict takes longer to detach because it retrieves the GUI state when it receives the detach request, while xmove collects the GUI state as it is created. To reduce our detach latency, we plan to enable the evict client library to incrementally fetch the font list during idle periods of the application process's execution, but we have not implemented this optimization yet. Guievict and xmove have similar re-attach performance,

Application	Guievict Latency (msec)		Xmove Latency (msec)	
	Detach	Re-attach	Detach	Re-attach
Xterm	21,042	46	32	134
Xmame	21,100	55	857	96
Emacs	21,198	148	45	230
Ghostview	21,655	379	315	307
Netscape	21,655	667	362	432

Table 1: Average detach and re-attach latency.

Stage	Time (usec)
Font List	21,052,039
Pixmap	562,205
Windows	31,824
Fonts	6,986
Graphics Contexts	1,977
Cursors	113
Colormaps	63

Table 2: Breakdown of detach latency for Netscape.

which is to be expected because they perform similar tasks during this stage.

4.2 Interactive Overhead

To measure the impact of guimux and xmove on interactive response we created a small application that repeatedly sends a request of minimal size (8 bytes) to the window server and waits for a reply of minimal size (32 bytes). We measured the average time for 1000 round trips for the application by itself, after it has been detached and re-attached with guievict, and through xmove. Our results are reported in Table 3.

Guievict and xmove have a measurable impact on the round trip time, caused by the overhead of redirecting the window system communication through a proxy. Since the overhead is less than a millisecond, it should not ever be perceptible to users.

System	Latency (usec)
None	70
Guievict	107
Xmove	133

Table 3: Average round trip time for a minimal X protocol request and reply.

5 SECURITY

Our system introduces three issues related to the security of X window applications and servers.

First, the owner of the application process must be able to control who is able to migrate or replicate its GUI. Our policy is to allow only the owner of a process to perform *guievict* operations on the process. This policy is enforced by two mechanisms. First, because the standard operating system protection prevents one user from modifying a process of another user, a process can only be hijacked by its owner. Second, the *evict* library authenticates the messages it receives from the *evictor*. It uses the credential passing mechanism of Unix domain sockets to ensure that the sender of a message is the same user who owns the application process. These mechanisms suffice to protect an application process from ordinary users, but not, of course, from the superuser of the execution host.

Second, the *guievict* server extension should not weaken the security of the X window system, a goal we believe we have met. Since the *GetResources* request only returns information about the resources of the application that issues the request, it cannot be used to learn about the resources of another application. Although a man-in-the-middle attack could be staged to inject a *GetResource* request in another application's connection to the window system, the information that would be revealed could also be obtained from passive eavesdropping on the connection, an old X windows vulnerability [8]. The defense, then and now, is to encrypt the connection.

Third, the owner of a desktop host must authorize *guievict* to re-attach a GUI to their display. Most X server access control mechanisms (such as MIT-MAGIC-COOKIE) require an authorized application to possess a server-generated capability that it can present to the server when it establishes a connection. This capability gives its possessor complete access to the X server. The desktop owner must have a secure way to transfer the capability to the *guievict* user, and they must trust the *guievict* user not to abuse the access to the server. *Guievict* does not provide capability transfer mechanisms and it does not change the access control policies of the X server. These issues are trivial in migration scenarios, where the desktop user and the *guievict* user are usually the same, since the user can transfer the capability when they log in to the execution host to run the *evictor*. However, these issues must be faced by a GUI replication system built over *guievict*'s replication mechanism.

6 RELATED WORK

Guievict most closely resembles *xmove* [22] in that both systems share the goal of migrating GUIs on a per-application basis from one desktop host to another. Unlike *guievict*, *xmove* requires the user to redirect in advance their application's window system connection through a proxy that tracks the state of its resources. In addition, *xmove* does not support the migration of application *processes*: it lacks a way to restore the communication between the application and proxy processes after the application process has migrated, as well as a way to migrate the proxy process or its state.

Other systems, including VNC [18], Teleporting [17,23], and Slim [20], provide remote access to a *session*, a collection of GUIs for remotely executing applications. Unlike *xmove*, these systems enable the user to migrate the display of all applications in the session as a single unit, a convenience for users who want remote access to their entire desktop, but a hindrance for users who want independent movement of their GUIs. Like *xmove*, these systems depend on a level of indirection that must be established when applications are started, and they do not support application process migration.

Recent systems have extended the session migration concept to include application process migration. Users of the Internet Suspend/Resume system [14] run their entire computing environment, from the operating system to their applications, inside a virtual machine whose state can be saved and regenerated on another machine. This system's mobility model is much coarser than *evict*'s: all applications (and their operating system) migrate as a single unit and GUI migration cannot be done separately from application process migration. The Zap system [16] provides a finer degree of mobility by allowing users to run their application processes in session abstractions that can be independently migrated to new hosts. However, Zap provides no support for GUI migration; its users must use systems like VNC to migrate their GUIs, adding another level of indirection.

Many systems have been developed to replicate the GUI of unmodified X windows applications [1,2,4,5,3,13] on multiple desktops. Like *xmove*, most of these systems require applications to be redirected to a proxy when they are started. One interesting exception is the HP Shared X [9] system, which performs replication through the use of an X server extension. Unlike the *guievict* extension, instead of providing a way to retrieve window session state from the server, the extended Shared X server itself acts a proxy that regenerates the GUI on the new displays and forwards messages between the application process and the new displays. The extension is thus unsuitable for GUI

migration because it does not allow the application to detach from its original X server.

7 CONCLUSION

Guievict enables the GUI of an ordinary X windows application to be migrated to another desktop host or replicated on multiple desktop hosts without premeditative steps such as redirecting the application process's communication through a proxy or relinking the application program binary. We have shown that server functionality necessary to retrieve a window session, a transportable representation of an application's GUI, is small and can be encapsulated in a window server extension without server recompilation, and that ordinary X windows applications can be hijacked at run time to retrieve their window session and perform GUI migration or replication.

We have implemented guievict for x86-based versions of Linux running the XFree86 window system. The code is freely available at <http://www.cs.wisc.edu/~zandy/guievict>.

REFERENCES

- [1] H.M. Abdel-Wahab and M.A. Feit. XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration. *IEEE TriCom '91: Communications for Distributed Applications and Systems*. Chapel Hill, NC, USA, April 1991, pp. 159-167.
- [2] H. Abdel-Wahab and K. Jeffay. Issues, Problems and Solutions in Sharing X Clients on Multiple Displays. *Internetworking - Research and Practice* 5, 1, March 1994, pp. 1-15.
- [3] J.E. Baldeschwieler, T. Gutekunst, B. Plattner. A Survey of X Protocol Multiplexors. *ACM SIGCOMM Computer Communication Review* 23, 2, April 1993.
- [4] J. Bazik. XMX - An X Protocol Multiplexor. <http://www.cs.brown.edu/software/xmx>.
- [5] C. Bormann and G. Hoffmann. Xmc and Xy - Scalable Window Sharing and Mobility. *8th Annual X Technical Conference*, January 1994.
- [6] B. Buck and J.K. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications* 14, 4, Winter 2000, pp. 317-329.
- [7] P. Chen and B. Noble. When Virtual is Better Than Real. *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau/Oberbayern, Germany, May 2001.
- [8] S. Garfinkel and G. Spafford. **Practical UNIX & Internet Security**, 2nd Edition. O'Reilly and Associates, Sebastopol, CA, April 1996.
- [9] D. Garfinkel, B.C. Welti, T.W. Yip. HP SharedX: A Tool for Real-Time Collaboration. *Hewlett-Packard Journal* 45, 2, April 1994, pp. 23-36.
- [10] J. Gettys. The Future is Coming: Where the X Window System Should Go. *2002 Usenix Annual Technical Conference (Freenix Track)*, Monterey, CA, June 2002, pp. 63-69.
- [11] J. Gettys and K. Packard. The X Resize and Rotate Extension - RandR. *2001 Usenix Annual Technical Conference (Freenix Track)*, Boston, MA, June 2001.
- [12] T. Gutekunst, D. Bauer, G. Caronni, Hasan, and B. Plattner. A Distributed and Policy-Free General-Purpose Shared Window System. *IEEE/ACM Transactions on Networking* 3, 1, February 1995.
- [13] O. Jones. Multidisplay Software in X: A Survey of Architectures. *The X Resource*, Issue 6, O'Reilly & Associates, Jan 1993, pp. 97-113.
- [14] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. *4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002)*, Callicoon, NY, June 2002, pp. 40-46.
- [15] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report #1346, Computer Sciences Department, University of Wisconsin, April 1997.
- [16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. *5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.
- [17] T. Richardson, F. Bennett, G. Mapp, and A. Hopper. Teleporting in an X Window System Environment. *The X Resource*, Issue 13, O'Reilly & Associates, Jan 1995, pp. 133-140.
- [18] T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing* 2, 1, January/February 1998, pp. 33-38.
- [19] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics* 5, 2, April 1986, pp. 79-109.
- [20] B.K. Schmidt, M.S. Lam, J.D. Northcutt. The Interactive Performance of SLIM: A Stateless, Thin-client Architecture. *17th ACM Symposium on Operating Systems Principles (SOSP '99)*. Kiawah Island, South Carolina, December 1999.
- [21] E. Solomita. Xmove Version 2.0 Beta 2. <ftp://ftp.cs.columbia.edu/pub/xmove>, November, 1997.
- [22] E. Solomita, J. Kempf and D. Duchamp. Xmove: A Pseudoserver for X Window Movement. *The X Resource*, Issue 11, July 1994, pp. 143-170.
- [23] K. Wood, T. Richardson, F. Bennett, A. Harter, and A. Hopper. Global Teleporting with Java: Towards Ubiquitous Personalized Computing. *Nomadics '96*, San Jose, March 1996.
- [24] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH Protocol Architecture. *Internet Engineering Task Force Internet-Draft draft-ietf-secsh-architecture-13*, September 2002.
- [25] V.C. Zandy, B.P. Miller, and M. Livny. Process Hijacking. *Eighth International Symposium on High Performance Distributed Computing (HPDC '99)*, Redondo Beach, CA, August 1999, pp. 177-184.
- [26] E.D. Zwicky, S. Cooper, and D.B. Chapman. **Building Internet Firewalls**, 2nd Edition. O'Reilly and Associates, Sebastopol, CA, June 2000.

CUP: Controlled Update Propagation in Peer-to-Peer Networks

Mema Roussopoulos Mary Baker
Department of Computer Science
Stanford University
{mema, mgbaker}@cs.stanford.edu

Abstract

This paper proposes CUP, a protocol for performing Controlled Update Propagation to maintain caches of metadata in peer-to-peer networks. To moderate propagation without imposing a global policy, CUP introduces the notion of individual node *investment return*. CUP allows each node to determine when it has economic incentive to receive and to propagate updates. A node participates in propagation only when the benefit (*investment return*) it secures from receiving and propagating updates outweighs its cost of propagation.

We extensively evaluate the CUP protocol in maintaining caches of metadata for locating content in peer-to-peer networks. We demonstrate that propagation of updates reduces the average latency of content search queries by as much as an order of magnitude across a variety of workloads. We propose and evaluate the use of *popularity-based* incentives to drive a node's propagation policy. These include incentives based on probabilistic as well as history-based models of investment return. Using these policies, we show that CUP nodes recover their propagation overhead by a factor of 2 to 300, thus offering a lean but powerful protocol.

1 Introduction

Peer-to-peer networks are self-organizing distributed systems where participating nodes both provide and receive services from each other in a cooperative effort without distinguished roles as pure clients or pure servers. Peer-to-peer networks have recently gained much attention, primarily because of the great number of features they offer applications that are built on top of them. These features include scalability, availability, fault tolerance, decentralized administration, and anonymity.

Along with these desirable features has come an array of technical challenges. For example, a fundamental problem in peer-to-peer systems is that of locating content. Given the name or a set of keyword attributes (metadata) of an object of interest, how do you locate the object within the peer-to-peer network? Most peer-to-peer networks return a set of metadata in response to a search query. This metadata typically consists of

index entries that point to the locations of nodes that serve replicas of the content of interest, but could also include other information such as pricing, trust, connection speed, or load information about these serving nodes.

Recent work suggests that metadata-based search queries for locating content can be a performance bottleneck in peer-to-peer systems [CRSB02]. As a result, designers of peer-to-peer systems suggest caching metadata at intermediate nodes that lie on the path taken by a search query [gnu, SBK02, RFH⁺01, SMK⁺01]. We refer to this as *Path Caching with Expiration* (PCX) because cached metadata entries typically have expiration times after which they are considered stale and require a new search.

PCX is desirable because it distributes query load for popular metadata items across multiple nodes, it reduces latency, and it alleviates hot spots. However, little attention has been given to how to *maintain* these intermediate caches. The cache maintenance problem is challenging because the peer-to-peer model assumes that the global set of valid metadata will change constantly as peer nodes join and leave the network, content is added to and deleted from the network, and replicas of existing content are added to alleviate bandwidth congestion at nodes holding the content. Nodes that cache metadata to serve queries in a more timely fashion need to know about changes to the metadata to serve queries better. Keeping cached metadata up-to-date therefore requires tracking which metadata items need to be updated, as well as tracking when interest in updating particular items at each cache has subsided to avoid unnecessary update propagation for the maintenance of these items.

In this paper we propose a protocol for performing Controlled Update Propagation (CUP) to maintain caches of metadata in a peer-to-peer network. CUP asynchronously builds caches of metadata while answering search queries. It then propagates updates of metadata to maintain these caches. To moderate this propagation, CUP introduces the notion of individual node *investment return*. Rather than imposing a global propagation policy, in CUP, nodes receive and propagate updates only when they have personal economic incentive

to do so. This occurs when the investment return (or benefit) a node secures by propagation outweighs the cost of propagation and thus, all overhead is recovered.

A node proactively receives updates for metadata items from a neighbor only if the node has registered interest with the neighbor. A node that proactively receives an update for a metadata item saves itself from handling a follow-up query for the same item that, without the application of the update, would otherwise miss at the node. Handling a miss involves generating network traffic to forward the query on to one's neighbor(s) and to receive a response. Therefore, from a node's perspective, a received update is *justified* if the update saves the node from the cost of handling queries. A node will only have interest in receiving updates as long as it continues to receive queries for that item.

In CUP, each node uses its own incentive-based policy to determine when to cut off its incoming supply of updates for an item. This way the propagation of updates is moderated and does not flood the network. We introduce several *popularity-based* incentives to drive a node's decisions to receive metadata updates. The first class of policies is probabilistic where a node computes the probability that a received update is justified using an estimate of the number of nodes that depend on this node for answers to queries for the item. The second class is "history-based," where the node compares the ratio of query arrivals to update arrivals in a sliding window of update arrivals. These policies favor the receipt of updates for popular items since these items generate queries most often.

Similarly, nodes decide individually when to propagate updates to interested neighbors. This is necessary because a node may not always be able or willing to forward updates to interested neighbors. In fact, a node's ability or willingness to propagate updates may vary with its workload. A salient feature of CUP is that even when a node's capacity to push updates becomes zero, nodes dependent on the node for updates fall back to the case of PCX and incur no overhead.

We compare CUP against PCX under typical workloads that have been observed in measurements of real peer-to-peer networks. We show that CUP reduces the average query latency by as much as an order of magnitude. CUP propagation overhead is more than compensated for by its savings in cache misses. The cost of saved misses can be two to 300 times the cost of updates pushed. Finally, since nodes make propagation decisions independently and without coordination from other nodes, CUP is simple to implement, which is crucial for a peer-to-peer network with potentially thousands of participants.

2 Background Terminology

The following terms give some background on how structured peer-to-peer networks perform their indexing and lookup operations. These help clarify the description of CUP over structured networks in the next section.

Node: This is a node in the peer-to-peer network. Each node periodically exchanges "keep-alive" messages with its neighbors to confirm their existence and to trigger recovery mechanisms should one of the neighbors fail.

Global Index: A fundamental operation in a peer-to-peer network is that of locating content. The basic idea in structured peer-to-peer networks is that a hashing scheme maps keys (names of content files or keywords) onto a virtual coordinate space using a uniform hash function that evenly distributes the keys to the space. The coordinate space serves as a global index that stores index entries which are (*key*, *value*) pairs. The value in an index entry is a pointer (typically an IP address) to the location of a node that stores a replica of the content associated with the entry's key. There can be several index entries for the same key, one for each replica of the content.

Authority Node: Each node *N* in a structured peer-to-peer system is dynamically allocated a subspace of the coordinate space (i.e., a partition of the global index) and all index entries mapped into its subspace are owned by *N*. We refer to *N* as the authority node of these entries. **Replicas** of content whose key corresponds to an authority node *N* send birth messages to *N* to announce they are willing to serve the content. Depending on the application supported, replicas might periodically send refresh messages to indicate they are still serving a piece of content. They might also send deletion messages that explicitly indicate they are no longer serving the content. These deletion messages notify the authority node to delete the corresponding index entry from its local index directory.

Local index directory: This is the subset of global index entries owned by a node.

Search Query: A search query posted at a node *N* is a request to locate a replica for key *K*. The response to such a search query is a set of index entries that point to replicas that serve the content associated with *K*.

Search/Routing Mechanism: In structured networks, when a node issues a query for key *K*, the query will be routed along a well-defined path with a bounded number of hops from the querying node to the authority node for *K*. The routing mechanism is designed so that each node on the path hashes *K* using the same hash function to deterministically choose which of its neighbors will serve as the next hop. The CUP protocol is aware of but neither affects nor is affected by the underlying routing mechanism.

Query Path for Key K : This is the path a search query for key K takes. Each hop on the query path is in the direction of the authority node that owns K . If an intermediate node on this path has unexpired entries cached, the path ends at the intermediate node; otherwise the path ends at the authority node. The reverse of this path is the **Reverse Query Path** for key K .

PCX: Recently, researchers have suggested caching metadata with expiration times along the reverse query path [gnu, SBK02, RFH⁺01, SMK⁺01] as the query response is propagated down to the querying node.

Cached index entries: This is the set of index entries cached by a node N in the process of passing up queries and propagating down query responses for keys for which N is not the authority. The set of cached index entries and the local index directory are disjoint sets.

Lifetime of index entries: Each index entry cached at a node has associated with it a lifetime during which it is considered fresh and after which it is considered expired.

3 CUP Protocol Design

We give a brief overview of CUP and then describe the components of the CUP protocol in detail.

3.1 CUP Overview

CUP is not tied to any particular search mechanism and therefore can be applied in both networks that perform structured search as well as networks that perform unstructured search. As described above, in structured search, queries follow a well-defined path from the querying node to an authority node that holds the index entries pertaining to the query [RFH⁺01, RD01a, SMK⁺01, ZKJ01]; in unstructured search, queries haphazardly travel through the network via flooding or random walks in search of index entries [gnu, LCC⁺02].

In the interest of space, in this paper we describe and evaluate how CUP works to maintain caches of index entries in structured peer-to-peer networks. The basic idea is that every node in the peer-to-peer network maintains two logical channels per neighbor: a query channel and an update channel. The query channel is used to forward search queries for objects of interest to the neighbor that is closest to the authority node holding the entries for those objects. The update channel is used to forward query responses asynchronously to a neighbor and to update index entries that are cached at the neighbor.

Queries for an item travel “up” the query channels of nodes along the path toward the authority node for that item. Updates travel “down” the update channels along the reverse path taken by a query. Figure 1 shows this process. The process of querying for items and updating cached index entries pertaining to those items forms a CUP tree, similar to an application-level multicast tree

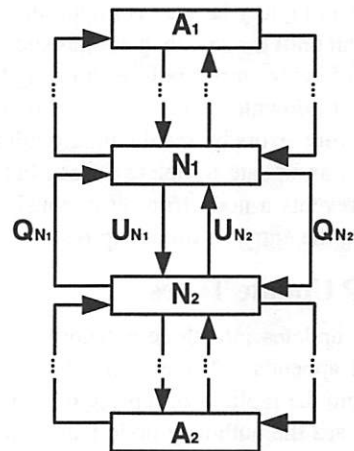


Figure 1: CUP Query & Update Channels. A_1 and A_2 are authority nodes for some objects. A query arriving at node N_2 for an item for which A_1 is the authority is pushed onto query channel Q_{N_1} to N_1 . If N_1 has a cached unexpired entry for the item, it returns it to N_2 through U_{N_1} . Otherwise, it forwards the query towards A_1 . Any update for an item originating from authority node A_1 flows downstream to N_1 which may forward it onto N_2 through U_{N_1} . The analogous process holds for queries at N_1 for items for which A_2 is one of the authority nodes.

where vertices are peer nodes interested in receiving updates for cached index entries.

The query channel enables “query coalescing”. If a node receives two or more queries for an item for which it does not have a fresh response, the node pushes only one instance of the query for that item up its query channel. This approach can have significant savings in traffic, because bursts of queries for an item are coalesced into a single request. Through simple bookkeeping (setting an interest bit) the node registers the interest of its neighbors so it knows which of its neighbors to push the query response to when it arrives.

The cascaded propagation of updates from authority nodes down the reverse paths of search queries has many advantages. First, updates extend the lifetime of cached entries allowing intermediate nodes to continue serving queries from their caches without re-issuing new queries. It has been shown that up to fifty percent of content hits at caches are instances where the content is valid but stale and therefore cannot be used without first being re-validated [CK01c]. These occurrences are called *freshness misses*. Second, a node that proactively pushes updates to interested neighbors reduces its load of queries generated by those neighbors. Third, the further down an update gets pushed, the shorter the distance subsequent queries need to travel to reach a fresh cached answer. As a result, search query latency is reduced.

Reducing search query latency is important because the user must wait until the search query has successfully returned a set of index entries before choosing from which replica node to download the content. Finally, updates can help prevent errors by invalidating outdated entries. For example, an update to delete a fresh but invalid index entry prevents a node from erroneously answering queries using the entry before it expires.

3.2 CUP Update Types

We classify updates into three categories: deletes, refreshes, and appends. Deletes, refreshes, and appends originate from the replicas of a piece of content and are directed toward the authority node that owns the index entries for that content.

Deletes are directives to remove a cached index entry. Deletes can be triggered by two events: 1) a replica sends a message indicating it no longer serves a piece of content to the authority node that owns the index entry pointing to that replica. 2) The authority node notices a replica has stopped sending “keep-alive” messages and assumes the replica has failed. In either case, the authority node deletes the corresponding index entry from its local index directory and propagates the delete to interested neighbors.

Refreshes are directive messages that extend the lifetimes of cached index entries. Refreshes that arrive at a cache do not prevent errors as deletes do, but help prevent freshness misses.

Finally, appends are directives to add index entries for new replicas of content. These updates help alleviate the demand for content from the existing set of replicas since they add to the number of replicas from which clients can download content.

3.3 CUP Node Bookkeeping

At each node, index entries are grouped together by key. For each key K , the node stores a “Pending-Response” flag that indicates whether the node is waiting to receive a response to a query for K , and an interest bit vector. Each bit in the vector corresponds to a neighbor and is set or clear depending on whether that neighbor is or is not interested in receiving updates for K .

Each node tracks the popularity or request frequency of each non-local key K for which it receives queries. The popularity measure for a key K can be the number of queries for K a node receives between arrivals of consecutive updates for K or a rate of queries in a sliding window of time. On an update arrival for K , a node uses its popularity measure to re-evaluate whether it is beneficial to continue caching and receiving updates for K . We elaborate on this cut-off decision in Section 4.4.

Node bookkeeping in CUP involves no network overhead and a few megabytes for hundreds of thousands of

entries. With increasing CPU speeds and memory sizes, this bookkeeping is negligible when we consider the reduction in query latency achieved.

3.4 Handling Queries in CUP

Upon receipt of a query for a key K , there are three basic cases to consider. In each of the cases, the node updates its popularity measure for K and sets the appropriate bit in the interest bit vector for K if the query originates from a neighbor. Otherwise, if the query is from a local client, the node maintains the connection until it can return a fresh answer to the client. To simplify the protocol description we use the phrase “push the query” to indicate that a node pushes a query upstream toward the authority node. We use the phrase “push the update” to indicate that a node pushes an update downstream in the direction of the reverse query path.

Case 1: Fresh Entries for key K are cached. The node uses its cached entries for K to push the response to the querying neighbor or local client.

Case 2: Key K is not in cache. The node adds K to its cache and marks it with a *Pending-Response* flag. The flag’s purpose is to coalesce bursts of queries for K into one query. A subsequent query for K will be suppressed since the node is already awaiting the response for the first query of the burst. Query coalescing results in significant network savings, for both PCX and CUP. In some of the workloads we evaluate, coalesced queries can form up to 90 percent of the total number of queries that miss.

With every query push, a timer is set so that if the query response is delayed, the node pushes up another query.

Case 3: All cached entries for key K have expired. The node must obtain the fresh index entries for K . If the *Pending-Response* flag is set, the node does not need to push the query; otherwise, the node sets the flag and pushes the query.

3.5 Handling Updates in CUP

A key feature of CUP is that a node does not forward an update for K to its neighbors unless those neighbors have registered interest in K . Therefore, with some light bookkeeping, CUP does not push unwanted updates.

Upon receipt of an update for key K there are three cases to consider.

Case 1: Pending-Response flag is set. This means that the update is a query response carrying a set of index entries in response to a query. The node stores the index entries in its cache, clears the *Pending-Response* flag, and pushes the update to neighbors whose interest bits are set and to local client connections open at the node.

Case 2: Pending-Response flag is clear. If all the interest bits for K are clear, the node decides whether

it wants to continue receiving updates for *K*. The node bases its decision on *K*'s popularity measure. Each node uses its own policy for deciding whether the popularity of a key is high enough to warrant receiving further updates for it. If the node decides *K*'s popularity is low, it pushes a *Clear-Bit* control message to the sender of the update to notify it that is no longer interested in *K*'s updates. Otherwise, if the popularity is high or some of the neighbor's interest bits are set, the node applies the update to its cache and pushes the update to those neighbors.

Note that a node can choose not to push updates for a key *K* to interested neighbors. This forces downstream nodes to fall back to PCX for *K*. However, by choosing to cut off downstream propagation, a node runs the risk of receiving subsequent queries from its neighbors which would cost it more, since it must both receive and respond to these queries. Therefore, although each node has the choice of stopping the update propagation at any time, it is in its best interest to push updates for which there are interested neighbors.

Case 3: Incoming update has expired. This could occur when the network path has long delays and the update does not arrive in time. The node does not apply the update and does not push it downstream. If the *Pending-Response* flag is set then the node re-issues another query for *K* and pushes it upstream.

3.6 Handling Clear-Bit Messages in CUP

A *Clear-Bit* control message is pushed by a node to indicate to its neighbor that it is no longer interested in receiving updates for a particular key from that neighbor.

When a node receives a *Clear-Bit* message for key *K*, it clears the interest bit for the neighbor from which the message was sent. If the node's popularity measure for *K* is low and all of its interest bits are clear, the node also pushes a *Clear-Bit* message for *K*. This propagation of *Clear-Bit* messages toward the authority node for *K* continues until a node is reached where the popularity of *K* is high or where at least one interest bit is set.

Clear-Bit messages can be piggybacked onto queries or updates intended for the neighbor, or if there are no pending queries or updates, they can be pushed separately.

3.7 Node Arrivals and Departures in CUP

The peer-to-peer model assumes that participating nodes will continuously join and leave the network. CUP must be able to handle both node arrivals and departures seamlessly.

Arrivals. When a new node *N* enters a structured peer-to-peer network, it becomes responsible for a portion of another node *M*'s share of the global index

and becomes the authority node for those index entries mapped into that portion. *N*, *M*, and all surrounding affected nodes (old neighbors of *M*) update the book-keeping structures they maintain for indexing and routing purposes. This is a necessary part of maintaining the connectivity of any structured peer-to-peer network when the set of nodes in the network changes.

For CUP, the issues at hand are updating the interest bit vectors of the affected nodes and deciding what to do with the index entries stored at *M*. This may require bit vector translation. For example, if a node that previously had *M* as its neighbor now has *N* as its neighbor, the node must make the bit ID that pointed to *M* now point to *N*.

To deal with its stored index entries, *M* could simply not hand over any of its entries to *N*. This would cause entries at some of *M*'s previous neighbors to expire and subsequent queries from those nodes would establish new update propagations from *N*. Alternatively, *M* could give a copy of its stored index entries to *N*. Both *N* and *M* would then go through each entry and patch their bit vectors. Both solutions are viable. The first solution requires no bit translation but temporarily loses the CUP update benefits and behaves like PCX for the untransferred entries. The second solution gets the CUP benefits for the transferred entries, at the expense of transferring them and performing the bit vector patching. The metadata and bit vectors for thousands of index entries can be compressed into a few kilobytes and can be piggybacked onto messages that are already being exchanged to reconfigure the topology. Once the transfer occurs, the bit vector patching is an in-memory, local operation that with today's CPU and memory capacities takes only a few seconds for a few million entries.

Departures. Node departures can be either graceful (planned) or ungraceful (due to sudden failure of a node). In either case the peer-to-peer index mechanism dictates that a neighboring node *M* take over the departing node *N*'s portion of the global index. To support CUP, the interest bit vectors of all affected nodes must be patched to reflect *N*'s departure.

If *N* leaves gracefully, *N* can choose not to hand over to *M* its index entries. Any entries at surrounding nodes that were dependent on *N* to be updated will simply expire and subsequent queries will establish new update propagations. Again, alternatively *N* may give *M* its set of entries. *M* must then merge its own set of index entries with *N*'s, by eliminating duplicate entries and patching the interest bit vectors as necessary. If *N*'s departure is due to a failure, there can be no hand-over of entries and all entries in the affected neighboring nodes will expire as in PCX.

4 Evaluation

The main goal of CUP is to continuously harvest the benefits of PCX. In doing so, there are two key performance questions to address. First, by how much does CUP reduce the average query latency? Second, how much overhead does CUP incur in providing this reduction?

We first define the notion of a CUP tree. We use this definition to present a cost model based on economic incentive used by each node to determine when to cut off the propagation of updates for a particular key. We give a simple analysis of how the cost per query is reduced (or eliminated) through CUP. We then describe our experimental results comparing the performance of CUP with that of PCX.

4.1 CUP Trees

Figure 2 shows a snapshot of CUP in progress for a network with seven peer nodes. The left half of each node shows the set of keys for which the node is the authority. The right half shows the set of keys for which the node has cached index entries as a result of handling queries. For example, node C owns K1 and K2 and has cached entries for K3, K4, and K5.

The process of querying for a key K and updating cached index entries pertaining to K forms a tree which we refer to as the *Real CUP Tree*. This tree, denoted $R(A, K)$, is similar to an application-level multicast tree and has as its root the authority node A for K. The exact structure of $R(A, K)$ depends on the actual workload of queries for K. The branches of the tree are formed by the paths traveled by queries from other nodes in the network. For example, in Figure 2, the tree $R(C, K1)$ has grown branch $\{F, D, C\}$ as the result of a query for K1 at node F. Updates for K1 originate at the root (authority node) C and travel down the tree to interested nodes A, D, E, and F. The entire workload of queries for all keys results in a collection of criss-crossing Real CUP Trees with overlapping branches.

We define the *Spanning CUP Tree* for key K, $S(A, K)$ as the tree that contains all possible query paths for K. This is the tree that would be generated by issuing a query for K from every node in the peer-to-peer network. For example, in Figure 2, $S(C, K1)$ is rooted at C (level 0), has nodes A, B, D, E at level 1, and nodes F and G at level 2.

4.2 Cost Model

Consider a node N within spanning tree $S(A, K)$ that is at distance D from A. We define the cost per query for K at N as the number of hops in the peer-to-peer network that must be traversed to return an answer to N. When a query for K is posted at N for the first time, it travels toward A. If none of the nodes between N and A have a

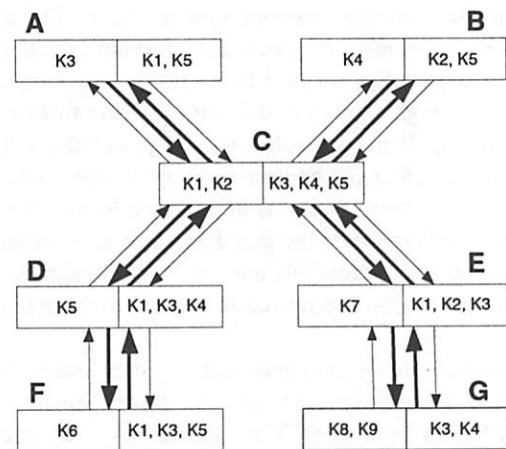


Figure 2: CUP Trees

fresh response cached, the cost of the query at N is $2D$: D hops up and D hops for the response to travel down. If a node on the query path has a fresh answer cached, the cost is less than $2D$. Subsequent queries for K at N that occur within the lifetime of the entries now cached at N have a cost of zero. As a result, caching at intermediate nodes can significantly lower average query latency.

We can gauge the performance of CUP by calculating the percentage of updates CUP propagates that are "justified", i.e., those whose cost is recovered by a subsequent query. Updates for popular keys are likely to be justified more often than updates for less popular keys.

A refresh update is justified if a query arrives sometime between the previous expiration of the cached entry and the new expiration time supplied by the refresh update. An append update is justified if at least one query arrives between the time the append is performed and the time of its expiration. Finally, a deletion update is justified if at least one query arrives between the time the deletion is performed and the expiration time of the entry to be deleted.

For each update, let T be the critical time interval described above during which a query must arrive in order for the update to be justified. Consider a node N at distance D from A in $R(A, K)$. An update propagated down to N is justified if at least one query is posted within T time units at any of the nodes of the spanning subtree $S(N, K)$. For example, if we assume a Poisson query arrival rate λ of one query per second at nodes in $S(N, K)$ and $T = 6$, then the probability that an update arriving at N is justified is $1 - e^{-\lambda T} = 1 - e^{-1 \cdot 6} = .99$.

The benefit of a justified CUP update goes beyond just recovery of its cost. For each hop a justified update u is pushed down to the root N of subtree $S(N, K)$, exactly one hop is saved since without u 's propagation, entries in all nodes of $S(N, K)$ will expire and the first subsequent query landing at a node N_i in $S(N, K)$ within T

time units will cause two hops, from N to its parent and back. This halves the number of hops traveled between N and its parent which in turn reduces query latency. In fact all subsequent queries posted somewhere in $S(N,K)$ within T time units will benefit from N receiving u . The cumulative benefit an update u brings to subtree $S(N,K)$ increases when N is closer to the authority node since there is a higher probability that queries will be posted within $S(N,K)$. We define “investment return” as the cumulative savings in hops achieved by pushing a justified update to node N . The experiments show that the return is large even when CUP’s reduction in latency is modest and is substantially large when the latency reduction is high.

4.3 Experiment Setup and Metrics

We evaluate CUP by comparing it with PCX with coalescing. We perform our simulation experiments using models derived from measurements of real peer-to-peer workloads [Mar02, SGG02, LCC⁺02, Sri01].

For our experiments, we simulate a content-addressable network (CAN) [RFH⁺01] using the Stanford Narses simulator [MGB01]. Again, we stress that CUP is independent of the specific search mechanism used by the peer-to-peer network and can be used as a cache maintenance protocol in any peer-to-peer network.

As in previous studies (e.g., [RFH⁺01, SMK⁺01, RD01b, CRSB02, RKCD01, RD01a, ZKJ01]), we measure CUP performance in terms of the number of hops traversed in the overlay network. *Miss cost* is the total number of hops incurred by all misses, i.e. freshness and first-time misses. CUP overhead is the total number of hops traveled by all updates sent downstream plus the total number of hops traveled by all clear-bit messages upstream. (We assume clear-bit messages are not piggybacked onto updates. This somewhat inflates the overhead measure.) *Total cost* is the sum of the *miss cost* and all overhead hops incurred. Note that in PCX, the *total cost* is equal to the *miss cost*. *Average query latency* is the average number of hops a query must travel to reach a fresh answer plus the number of hops the answer must travel downstream to reach the node where the query was posted. For coalesced queries, we count the number of hops each coalesced query waits until the answer arrives. Thus, the average latency is over all queries, including hits, coalesced misses and non-coalesced misses.

We compute investment return (IR) as the overall ratio of saved miss cost to overhead incurred by CUP:

$$IR = \frac{MissCost_{PCX} - MissCost_{CUP}}{OverheadCost_{CUP}}$$

Thus, as long as IR is greater than or equal to 1, CUP fully recovers its cost.

The simulation takes as input the number of nodes in the overlay peer-to-peer network, the number of keys owned per node, the distribution of queries for keys, the distribution of query inter-arrival times, the number of replicas per key, the lifetime of index entries in the system, and the fraction of an entry’s lifetime remaining at which refreshes for the entry are pushed out from the authority node. We present experiments for $n = 2^k$ nodes where k ranges from 7 to 14. After a warm-up period for allowing the peer-to-peer network to connect, the measured simulation time is 3000 seconds. Since both Poisson and Pareto query inter-arrival distributions have been observed in peer-to-peer environments [LCC⁺02, Mar02], we present experiments for both distributions. Nodes are randomly selected to post queries. We also performed experiments where queries are posted at particular “hot spots” in the network and found similar results. These, as well as other results which we omit in the interest of space, can be found elsewhere [Rou02].

We present results for experiments where index entry lifetimes are five minutes and refreshes occur one minute before expiration. We choose these values to reflect the dynamic and unpredictable nature of peer-to-peer networks. It has been found that the median user session duration of a peer is approximately sixty minutes [SGG02]. However, content may become available on a peer or be deleted from the peer at any point during the user session. This results in actual content availability that is on the order of a few minutes [CLL02]. We therefore take the safe approach of validating that the content is still available every few minutes. This is also in line with designers of structured peer-to-peer networks who advocate periodic refreshes (keep-alive messages) between the peers storing replicas of a particular content and the authority node for that content [RFH⁺01, RD01a]. If there were some way to ensure that lifetimes of entries could be set for longer, then we find that CUP continues to provide benefits, albeit reduced, since PCX would incur fewer misses. Unfortunately, making such guarantees would require placing a global availability policy across autonomous peer nodes.

We present six sets of experiments. First, we compare the effect on CUP performance of different incentive-based cut-off policies and compare the performance of these policies to that of PCX. Second, using the best cut-off policy of the first experiment, we study how CUP performs as we scale the network. Third, we study the effect on CUP performance of varying the topology of the network by increasing the average node degree, thus decreasing the diameter of the network. Fourth, we study the effect on CUP performance of limiting the outgoing update capacities of nodes. Fifth, we study how CUP performs when queries arrive in bursts, as observed

Table 1: Total cost per key per query rate for varying cut-off policies.

Policy	1 q/s Total Cost	10 q/s Total Cost	100 q/s Total Cost	1000 q/s Total Cost
PCX	61568 (1.00)	154502 (1.00)	476420 (1.00)	2296869 (1.00)
Linear, $\alpha = 0.25$	55475 (0.90)	72022 (0.47)	49341 (0.10)	196650 (0.09)
Linear, $\alpha = 0.10$	41281 (0.67)	34311 (0.22)	47132 (0.10)	196650 (0.09)
Logarithmic, $\alpha = 0.5$	31658 (0.51)	27311 (0.18)	47785 (0.10)	196797 (0.09)
Logarithmic, $\alpha = 0.25$	30683 (0.50)	24695 (0.16)	48330 (0.10)	196797 (0.09)
Second-chance	16958 (0.28)	23702 (0.15)	48330 (0.10)	196797 (0.09)
Optimal push level	15746 (0.26)	23696 (0.15)	45325 (0.095)	153309 (0.07)

with Pareto inter-arrivals. These five experiments show the per-key benefits of CUP when keys are queried for according to a uniform distribution. In the last experiment, we show the overall benefits of CUP when keys are queried for according to a Zipf-like distribution.

4.4 Varying the Cut-Off Policies

As discussed in Section 4.2, the propagation of updates is beneficial only if the updates are justified; when a node's incentive to receive updates for a particular key fades, continuing update propagation to that node simply wastes network bandwidth. Therefore, each node needs an independent and decentralized way of controlling its intake of updates.

We base a node's incentive to receive updates for a key on the *popularity* of the key at the node. The more popular a key is, the more incentive there is to receive updates for that key, because updates for that key are more likely to be justified. For a key K , the popularity is the number of queries a node has received for K since the last update for K arrived at the node. (Note that the popularity metric is node-dependent and could be defined in another way such as with a moving average of query arrivals for K .)

We examine two types of thresholds against which to test a key's popularity when making the cut-off decision: probability-based and history-based.

A probability-based threshold uses the distance of a node N from the authority node A to approximate the probability that an update pushed to N is justified. Per our cost model of section 4.2, the further N is from A , the less likely an update at N will be justified. We examine two such thresholds, a linear one and a logarithmic one. With a linear threshold, if an update for key K arrives at a node at distance D and the node has received at least αD queries for K since the last update for some constant $\alpha \geq 0$, then K is considered popular and the node continues to receive updates for K . Otherwise, the node cuts off its intake of updates for K by pushing up a clear-bit message. The logarithmic popularity threshold is similar. A key K is popular if the node has received $\alpha \lg(D)$ queries since the last update. The logarithmic threshold is more lenient than the linear in that it increases at a slower rate as we move away from the root.

A history-based threshold is one that is based on the recent history of the last n update arrivals at the node. If within n updates, the node has not received any queries, then the key is not popular and the node pushes up a clear-bit message. A specific example of a history-based policy is the "second-chance policy", $n = 2$. When an update arrives, if no queries have arrived since the last update, the policy gives the key a "second chance" and waits for the next update. If at the next update, still no queries for K have been received, the node pushes a clear-bit message. The philosophy behind this policy is that pushing these two updates down from the node's parent costs the same as one query miss occurring at the node, since a query miss incurs one hop up to the parent and one hop down. This means that just one query arriving at the node between the first update and the expiration of the second update is enough to recover their propagation cost.

Table 1 compares PCX with CUP using the linear and logarithmic policies for various α values, with CUP using second chance, and with a version of CUP that does not use any cut-off policy but instead pushes updates until the optimal push level is reached. To determine the optimal push level we make CUP propagate updates to all querying nodes that are at most p hops from the authority node. By varying the push level p , we determine the level which achieves minimum total cost. This is shown by the row labeled "optimal push level" and used as a baseline against which to compare PCX and CUP with the cut-off policies described.

In Table 1 we show the cut-off policy results for a network of 1024 nodes and Poisson λ rates of 1, 10, 100 and 1000 queries per second. In each table entry, the first number is the total cost and the number in parentheses is the total cost normalized by the total cost for PCX. First, we see that regardless of the cut-off policy used, CUP outperforms PCX. Second, for the lower query rates, the performance of the linear and the logarithmic policies is greatly affected by the choice of parameter α , whereas for the higher query rates, the choice of α is less dramatic. These results show that choosing a priori an α value for the linear and logarithmic policies that will perform well across all workloads is difficult.

For the higher query rates, the history-based second-

Table 2: Per-Key Comparison of CUP with PCX for varying network sizes, Poisson arrivals of 1 query/second.

Network Size	128	256	512	1024	2048	4096	8192	16384
CUP/PCX MissCost	0.10	0.10	0.15	0.17	0.19	0.22	0.20	0.21
PCX AvgLat (σ)	1.51 (2.77)	2.67 (3.96)	4.49 (5.92)	6.74 (8.25)	11.01 (12.11)	17.47 (17.49)	29.29 (27.79)	45.56 (40.31)
CUP AvgLat (σ)	0.21 (1.10)	0.46 (1.60)	1.25 (3.19)	2.17 (4.37)	4.18 (7.13)	7.70 (11.28)	11.48 (15.08)	19.17 (23.75)
IR/CUPOvhd Hop	4.15	4.88	6.29	7.83	11.43	16.14	24.85	35.98

chance policy performs comparably to the probability-based policies, and for the lower query rates outperforms the probability-based policies. In fact, across all rates, the second-chance policy achieves a total cost very near the optimal push level total cost. In all remaining experiments, we use second-chance as the cut-off policy.

4.5 Scaling the Network

In this section we study CUP performance as we scale the size of the network.

Table 2 compares CUP and PCX for network sizes between $2^7 = 128$ and $2^{14} = 16384$ nodes for a Poisson λ rate of 1 query per second. The first row shows the CUP miss cost as a fraction of the PCX miss cost. The second and third rows show the average query latency in hops for PCX and CUP respectively. The number in parentheses is the standard deviation. As can be observed, CUP reduces average query latency respectively by 9.77, and 17.81, and 26.39 hops for the 4096, 8192, and 16384 node networks. This is a substantial reduction in average query latency that improves with increasing network size. Comparing the standard deviations of CUP and PCX we see that CUP also has less variability around its average query latency.

The fourth row in Table 2 shows the IR per overhead push performed by CUP. We observe a growth in the rate of return with 16.14, 24.85, and 35.98 for the last three network sizes. These numbers are quite strong, considering that the overhead is completely recovered.

Figure 3 shows the IR of CUP versus network size for Poisson with $\lambda = 1, 10, 100$, and 1000 queries per second. From the figure we see that for a particular network size, if we increase the query rate the IR increases, and for a particular query rate, if we increase the network size, the IR also increases. This demonstrates that CUP scales to higher query rates and higher network sizes.

4.6 Varying the Network Topology

In general, different peer-to-peer networks exhibit different topologies and thus different network diameters. The particular topology created depends on the protocol the peer nodes use to join the network and to keep it connected. The CAN design is based on a d -dimensional coordinate space, with our experiments thus far having been for $d = 2$. Increasing the number of dimensions results in a topology where nodes have higher degree

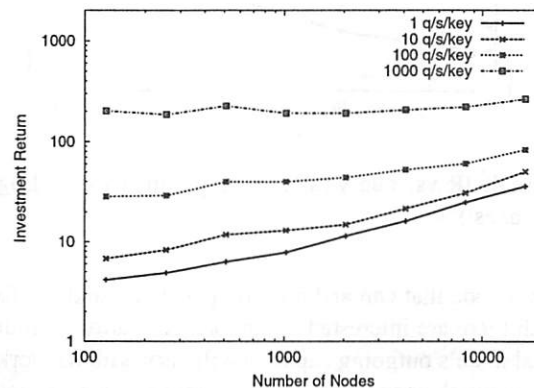


Figure 3: IR vs. net size. (Log-scale axes.)

and the network has smaller diameter. Smaller diameter means that the average path length of a query on a miss is shorter for both PCX and CUP, which implies that the benefits of CUP may be less pronounced. On the other hand, CUP total update cost also decreases since there will be shorter distances for updates to travel. As a result, we find that CUP continues to provide significant savings in terms of both overall total cost, latency reduction, and IR per overhead push.

In this set of experiments we study the effect of increasing the number of CAN dimensions on a network with 1024 nodes. The dimensions chosen for this experiment are 2, 3, 5, and 10. These dimensions result in network diameters of 24, 12, 8, and 8 respectively. (For a network of 1024 nodes, increasing beyond five dimensions does not reduce the network diameter any further.) The queries arrive according to a Poisson process with λ rate of 1, 10, 100, and 1000 queries per second. Figure 4 shows the IR versus the query rate for each dimension. From the figure we see that the curves for dimensions 5 and 10 are very similar because they have equal network diameters. We also see that dimension 2 achieves the highest IR across all query rates, and that the IR decreases with dimension. However, even for the higher dimensions (5 and 10), the IR is at least 2.1 for 1 q/s and increases to 36.6 for 1000 q/s.

4.7 Varying Outgoing Update Capacity

Our experiments thus far show that CUP outperforms PCX under conditions where all nodes have full outgoing update capacity. A node with full outgoing capac-

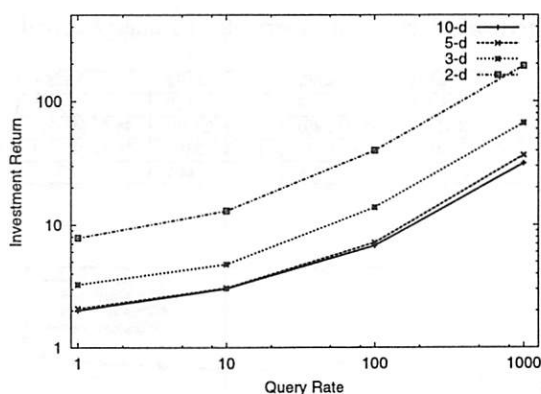


Figure 4: IR vs. query rate, varying dimensions. (Log-scale axes.)

ity is a node that can and does propagate all updates for which there are interested neighbors. In reality, an individual node's outgoing capacity will vary with its workload, network connectivity, and willingness to propagate updates. In this section we study the effect on CUP performance of reducing the outgoing update capacity of nodes.

We present an experiment run on a network of 1024 nodes. In this experiment, after a five minute warm up period, we randomly select twenty percent of the nodes and reduce their outgoing capacity to a fraction of their full capacity. These nodes operate at reduced capacity for ten minutes after which they return to full capacity. After another five minutes for stabilization, we randomly select another set of twenty percent of the nodes and reduce their capacity for ten minutes. We proceed this way for the entire 3000 seconds during which queries are posted, so capacity loss occurs three times during the simulation.

Figure 5 shows the ratio of CUP total cost to PCX total cost versus capacity c for this experiment and for four different Poisson query rates λ . The capacity c ranges from 0, implying that no updates are propagated, to 1, where nodes have full outgoing capacity. $c = .25$ means that a node is only capable/willing of pushing out one-fourth the updates it receives.

Note that even when one fifth of the nodes do not propagate any updates, the total cost incurred by CUP is about half that of PCX. As the outgoing capacity increases, the total cost decreases smoothly until $c = 1$ where CUP achieves its full potential. A key observation from these experiments is that CUP's performance degrades gracefully as the capacity c decreases. This is because reduction in update propagation also results in reduction of its associated overhead. Therefore, the capacity reduction should be seen as a missed opportunity for higher returns rather than as an overall loss. Clearly

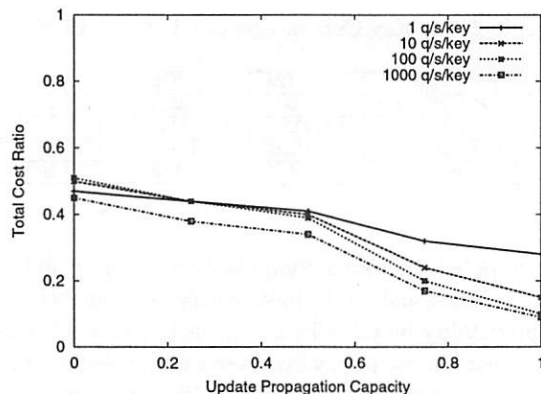


Figure 5: Total cost ratio vs. update propagation capacity

though, CUP achieves its full potential when all nodes have maximum propagation capacity.

4.8 Pareto Query Arrivals

Recent work has observed that in some peer-to-peer networks, query inter-arrivals exhibit burstiness on several time scales [Mar02], making the Pareto distribution a good candidate for modeling these inter-arrival times. Therefore, in this section we compare CUP with PCX under Pareto inter-arrivals.

The Pareto distribution has two parameters associated with it: the shape parameter $\alpha > 0$ and the scale parameter $\kappa > 0$. The cumulative distribution function of inter-arrival time durations is $F(x) = 1 - (\frac{\kappa}{x+\kappa})^\alpha$. This distribution is heavy-tailed with unbounded variance when $\alpha < 2$. For $\alpha > 1$, the average number of query arrivals per time unit is equal to $\frac{(\alpha-1)}{\kappa}$. For $\alpha \leq 1$, the expectation of an inter-arrival duration is unbounded and therefore the average number of query arrivals per time unit is 0.

We ran experiments for a range of α and κ values but can only present representative results here. Table 3 compares CUP with PCX for α equal to 1.25 and 1.1 respectively for a network of 1024 nodes. We set the value of κ in each run so that the average rate of arrivals $\frac{(\alpha-1)}{\kappa}$ equals 1, 10, 100, and 1000 queries per second to match the λ rate of the Poisson experiments in previous sections.

As α decreases toward 1, query interarrivals become more bursty. Queries arrive in more frequent and more intense bursts, followed by idle periods of varying lengths. If an idle period occasionally falls in the heavy-tail portion of the Pareto distribution (i.e., it is a very long idle period), then second chance CUP propagation cost could be unrecoverable, since the next query may arrive long after the cached entry has expired. However, CUP does well under bursty conditions because when

Table 3: Per-Key, Per-Query Rate Comparison of CUP with PCX for Pareto arrivals.

Average Rate (q/s)	1	1	10	10	100	100	1000	1000
Pareto rate (α)	1.25	1.1	1.25	1.1	1.25	1.1	1.25	1.1
CUP/PCX MissCost	0.24	0.14	0.08	0.07	0.07	0.09	0.08	0.08
PCX AvgLat (σ)	7.77 (9.28)	6.99 (9.43)	3.84 (8.41)	4.01 (8.75)	1.75 (5.88)	1.61 (5.53)	1.00 (4.02)	1.10 (4.16)
CUP AvgLat (σ)	3.16 (5.75)	1.71 (4.44)	0.42 (3.03)	0.37 (2.80)	0.13 (1.66)	0.15 (1.71)	0.08 (1.17)	0.09 (1.24)
IR/CUPovhd Hop	6.41	7.49	13.09	16.03	43.25	53.57	223.97	293.30

it is able to refresh a cache before a burst of queries, it saves a large penalty which by far outweighs any unrecovered overhead that occurs during the occasional, very long idle period. Therefore, refreshing the cache in time provides greater benefits with increasing burstiness. The table results confirm this. In going from $\alpha = 1.25$ to $\alpha = 1.1$, we see that the average query latency reduction CUP achieves generally improves and the IR increases for all query rates.

4.9 Zipf-like Key Distributions

A recent study has shown that queries for multiple keys in a peer-to-peer network follow a Zipf-like distribution, with a small portion of the keys getting the most queries [Sri01]. That is, the number of queries received by the i 'th most popular key is proportional to $\frac{1}{i^\alpha}$ for constant α .

In this section we compare CUP with PCX in a network of 1024 nodes, where each node owns one key. The query distribution among the 1024 keys follows a Zipf-like distribution with parameter $\alpha = 1.2$. Table 4 shows results for Poisson arrivals where the overall λ rates are 100, 1000, 10000, and 100000 queries per second. (We also ran experiments with $\alpha = 0.80$ and 2.40 and with Pareto arrivals, and the results were similar.)

From the table we see that CUP outperforms PCX with IR ranging from 6.57 to 30.02. The latency reduction ranges from 3.2 (for 100 q/s) to an order of magnitude reduction (for 100000 q/s, latency dropped from 1.53 to 0.13). The Zipf-like distribution causes some of the keys to get a large percentage of the queries, leaving others to be asked for quite rarely. For rare keys, caching does not help since the entry expires by the time the key is queried for again, and the query rate for these keys is not high enough to recover the update propagation. However, the IR for the very hot keys is high enough to by far offset the unrecovered cost of the unpopular keys. As a result, CUP achieves an overall IR of at least 6.57 for 100 q/s and as much as 30.02 for 100000 q/s.

5 Related Work

We describe related work specifically in the peer-to-peer literature, followed by related work in the systems literature in general.

5.1 Related Peer-to-Peer Work

To our knowledge, CUP is the first protocol aimed at maintaining caches of index entries to improve search queries in peer-to-peer networks. While designers of peer-to-peer systems advocate caching index entries to improve performance [gnu, RFH⁺01, SMK⁺01, RD01a], there has been little follow-up work studying when and where to cache entries and how to maintain these cached entries in a peer-to-peer system.

Cox et al. [CMM02] study providing DNS service over a peer-to-peer network as an alternative to traditional DNS. They cache index entries, which are DNS mappings, along search query paths. Similarly, the TerraDir Distributed Directory caching scheme [SBK02] has nodes along the search query path cache pointers to other nodes previously traversed by the query. In each of these examples, cached index entries have expiration times and are not refreshed or maintained until a miss or failure occurs.

Path caching of content in peer-to-peer systems has received more attention. Freenet [CSWH00], CFS [DKK⁺01], PAST [RD01b], and Lv et al. [LCC⁺02] each perform path caching, or caching of content along the search path of a query. These studies do not focus on cache maintenance, but rather depend on expiration or cache size constraints to implicitly prevent the use of stale content.

CUP trees are similar to application-level multicast trees, particularly those built on peer-to-peer networks. These include Scribe [RKCD01] and Bayeaux [ZZJ⁺01]. Scribe is a publish-subscribe infrastructure built on top of Pastry [RD01a] where subscribers interested in a topic join its corresponding multicast group. Scribe creates a multicast tree rooted at the rendez-vous point of each multicast group. Publishers send a message to the rendez-vous point which then transmits the message to the entire group by sending it down the multicast tree. The multicast tree is formed by joining the Pastry routes from each subscriber node to the rendez-vous point. Scribe could benefit from our CUP ideas to provide update propagation for cache maintenance in Pastry.

Table 4: Cross-Key Comparison of CUP with PCX, for Poisson arrivals and Zipf-like key distribution

Overall AvgRate q/s	100	1000	10000	100000
CUP/PCX MissCost	0.45	0.23	0.10	0.08
PCX AvgLat (σ)	10.6 (9.9)	6.9 (8.9)	3.4 (7.5)	1.53 (5.47)
CUP AvgLat (σ)	7.4 (8.5)	2.6 (5.2)	0.4 (2.7)	0.13 (1.67)
IR	6.57	8.52	10.98	30.02

5.2 Related Distributed Caching Work

DNS [Moc87a, Moc87b] is the largest and best known distributed directory service for the Internet. Name servers, like CUP nodes, can be viewed as distributed caches that hold index entries (DNS name-to-IP address mappings) with Time-to-Live (TTL) fields indicating how long they should be considered valid. The maintenance of DNS caches has typically been *pull-driven*, where name servers either pull a fresh version of a stale cached mapping in response to a client request, or proactively, in anticipation of a request [CK01b]. CUP maintains caches through a *proactive push-driven* approach, where updates are pushed to all interested nodes in the overlay network. DNS is generally intended to support slowly-changing mappings with TTLs on the order of hours (e.g., 24 hours) [CK01b], whereas CUP is geared toward maintaining caches of metadata that change frequently, on the order of minutes.

Distributed caching techniques have been looked at in the context of distributed file systems (e.g., [HO93, ADN⁺95]), where the focus is on achieving cache coherence amongst groups of participating file writers that have cached files and communicate over a local-area network. CUP is designed for peer-to-peer environments, where there may be thousands of participating nodes spread across the Internet, and where updates for a particular metadata item are typically generated by only one peer node.

Distributed caching techniques have also been looked at in the context of web caching. Many previous studies have focused on cache replacement policies since cache size becomes a finite source when caching content for potentially thousands of clients [Mog96, WAS⁺96]. In CUP, cache size is not an issue since metadata are small.

Data caching and movement techniques based on economic models of locally computed interest have been studied in the context of the Mariposa Distributed Database Management System [SDK⁺94]. Mariposa builds a market-based system with a virtual currency where servers advertise prices to provide resources such as CPU cycles and storage services for query processing such that they maximize their local revenue income per time unit. If a server is underutilized, it will lower the price of its resources to attract more requests. In CUP, the notion of economic benefit is different; a node that derives benefit by propagating an update is saving itself

from future work (query requests).

Many schemes have been proposed for the maintenance of cached web content. Some propose push-based invalidation schemes where a web server/proxy notifies proxies/clients when cached objects are modified (e.g., [LC99]), pull-based validation schemes, where the proxy/client validates with the server/proxy cached objects that have expired [CK01c], and hybrid schemes, where the server piggybacks validations on responses to requests for related objects (e.g., [KW97]). CUP differs from previous web maintenance schemes by using push-driven propagation that is driven by the individual economic incentive of participating nodes.

Cooperative caching has been proposed to allow groups of participating caches to exchange cached web content amongst themselves. The overall goal is to bring a particular web object to the cache that is closest to the clients requesting that web object. Previous proposals include hierarchical cache schemes (e.g., [CDN⁺96, KLL⁺97, squ, CK01a]), hash-based schemes [KLL⁺97, VR98], directory-based schemes [FCAB98, MIB98, TDVK99], and multicast-based schemes (e.g., [Tou98]). Of these cooperative caching studies, those most related to CUP are work on refreshment policies for cascaded caches by Cohen et al. [CK01a] and work on distributing location hints across a hierarchy of caches by Tewari et al. [TDVK99].

Cohen and Kaplan study the effect that aging through cascaded caches has on the miss rates of web client caches [CK01a]. For each object an intermediate cache refreshes its copy of the object when its age exceeds a fraction ν of the lifetime duration. The intermediate cache does not push this refresh to the client cache; instead, the client cache waits until its own copy has expired at which point it fetches the intermediate cache's copy with the remaining lifetime. For some sequences of requests at the client cache and some ν 's, the client cache can suffer from a higher miss rate than if the intermediate cache only refreshed on expiration. A CUP tree could be viewed as a series of cascaded caches in that each node depends on the previous node in the tree for updates to an index entry. The key difference is that in CUP, refreshes are pushed down the entire tree of interested nodes. Therefore, whenever a parent cache gets a refresh so does the interested child node. In such situations, we find the miss rate at the child node actually

improves.

Tewari et al. [TDVK99] cache location hints in addition to web content at web caches in a web cache hierarchy. Location hints are used by requesting leaf caches to access copies of web content directly from remote caches holding the content, rather than waiting for the content to travel through the root and down to them. Propagation of hint updates is considered inexpensive, and occurs proactively and independently of the request pattern of the web object the hint represents. CUP emphasizes *recovering* propagation overhead. CUP makes the propagation decision by comparing the cost of propagating a particular update with the benefit (investment return) the update will bring to the tree below the node. CUP only propagates updates that are likely to benefit subsequent queries in the subtree below.

6 Conclusions

CUP provides a general purpose framework for maintaining caches of metadata in peer-to-peer networks, where continuous updates are expected, yet nodes must have personal economic incentive to participate in the maintenance. CUP is a complete protocol with query channels for coalescing bursts of queries and update channels for asynchronous delivery of query responses and updates of cached metadata. To moderate propagation without imposing a global policy, CUP introduces the notion of *investment return* for motivating each node to participate in the update propagation and policies for estimating when the benefit ceases to outweigh the overhead. For the case of locating content in a peer-to-peer network, we find that CUP secures an investment return of 2 to 300 times the propagation cost and significantly reduces query latency.

We have leveraged the CUP protocol to deliver metadata required for effective load-balancing of content downloads across multiple replica nodes [Rou02]. As with regular searches, the economic incentive-based model helps to moderate and control the amount of metadata update propagation in a highly dynamic environment where load information changes very rapidly. Future work includes the use of CUP to enhance management of dynamic content replication, publish-subscribe applications, and price negotiation and auctioning of services amongst nodes in a peer-to-peer network.

7 Acknowledgments

This research is supported by the Stanford Networking Research Center, and by DARPA (contract N66001-00-C-8015).

We thank Brian Noble, our paper shepherd, for his guidance. The work presented here has benefitted greatly from discussions with Petros Maniatis, Armando

Fox, Nick McKeown, and Rajeev Motwani. We thank them for their invaluable feedback.

References

- [ADN⁺95] T. Anderson, M. Dahlin, J. Neeffe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *SOSP*, 1995.
- [CDN⁺96] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell. A Hierarchical Internet Object Cache. In *USENIX*, January 1996.
- [CK01a] E. Cohen and H. Kaplan. Aging Through Cascaded Caches: Performance Issues in the Distribution of Web Content. In *Sigcomm*, 2001.
- [CK01b] E. Cohen and H. Kaplan. Proactive caching of DNS records: Addressing a performance bottleneck. In *SAINT*, 2001.
- [CK01c] E. Cohen and H. Kaplan. Refreshment Policies for Web Content Caches. In *Infocom*, 2001.
- [CLL02] J. Chu, K. Labonte, and B. N. Levine. Availability and Locality Measurements of Peer-to-Peer File Systems. In *Proc. ITCOM: Scalability and Traffic Control in IP Networks II Conferences*, July 2002.
- [CMM02] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *IPTPS*, March 2002.
- [CRSB02] Y. Chawathe, S. Ratnasamy, S. Shenker, and L. Breslau. Can Heterogeneity Make Gnutella Scale? May 2002. <http://research.att.com/yatin/publications/>.
- [CSWH00] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *DIAU*, July 2000.
- [DKK⁺01] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *SOSP*, 2001.
- [FCAB98] L. Fan, P. Cao, J. Almeida, and A.Z. Broder. Summary Cache: A scalable Wide-area Web Cache Sharing Protocol. In *SIGCOMM*, 1998.
- [gnu] The Gnutella Protocol Specification v0.4. <http://gnutella.wego.com/>.
- [HO93] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. In *SOSP*, 1993.
- [KLL⁺97] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, 1997.
- [KW97] B. Krishnamurthy and C.E. Wills. Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web. In *USITS*, 1997.
- [LC99] Dan Li and David Cheriton. Scalable Web Caching of Frequently Updated Objects Using Reliable Multicast. In *USITS*, 1999.
- [LCC⁺02] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured P2P Networks. In *ICS*, 2002.

- [Mar02] E. P. Markatos. Tracing a large-scale Peer-to-Peer System: an hour in the life of Gnutella. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.
- [MGB01] P. Maniatis, T.J. Giuli, and M. Baker. Enabling the Long-Term Archival of Signed Documents through Time Stamping. Technical Report cs.DC/0106058, Stanford University, June 2001.
- [MIB98] J.M. Menaud, V. Issarny, and M. Banatre. A New Protocol for Efficient Transversal Web Caching. In *Symposium on Distributed Computing*, 1998.
- [Moc87a] P. Mockapetris. Domain names - Concept and Facilities. In *RFC 1034*, 1987.
- [Moc87b] P. Mockapetris. Domain names - Implementation and Practice. In *RFC 1035*, 1987.
- [Mog96] J. Mogul. Hinted Caching in the Web. In *SIGOPS*, 1996.
- [RD01a] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *MiddleWare*, November 2001.
- [RD01b] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility". In *SOSP*, October 2001.
- [RFH⁺01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Sigcomm*, 2001.
- [RKCD01] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *NGC*, 2001.
- [Rou02] M. Roussopoulos. *Controlled Update Propagation in Peer-to-Peer Networks*. PhD thesis, Stanford University, 2002.
- [SBK02] B. Silaghi, B. Bhattacharjee, and P. Keleher. Routing in the TerraDir Directory Service, 2002. <http://motefs.cs.umd.edu/terradir/>.
- [SDK⁺94] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staeline. An Economic Paradigm for Query Processing and Data Migration in Mariposa. In *3rd International Conference on Parallel and Distributed Information Systems*, 1994.
- [SGG02] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *MMCN*, 2002.
- [SMK⁺01] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Sigcomm*, 2001.
- [squ] Squid Internet Object Cache. <http://squid.nlanr.net>.
- [Sri01] K. Sripanidkulchai. The Popularity of Gnutella Queries and its Implication on Scalability, February 2001. <http://www-2.cs.cmu.edu/kunwadee/research/p2p/gnutella.html>.
- [TDVK99] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *ICDCS*, 1999.
- [Tou98] J. Touch. The LSAM Proxy Cache - A Multicast Distributed Virtual Cache. In *3rd WWW Caching Workshop*, June 1998.
- [VR98] V. Valloppilli and K.W. Ross. Cache Array Routing Protocol v1.0 (Work in Progress), February 1998. <ftp://ftp.isi.edu/internet-drafts/draft-vinod-carp-v1-02.txt>.
- [WAS⁺96] S. Williams, M. Abrams, C.R. Standbridge, G. Abdulla, and E.A. Fox. Removal Policies in Network Caches for World-Wide Web Document. In *Sigcomm*, 1996.
- [ZKJ01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, April 2001.
- [ZZJ⁺01] S.Q. Zhuang, B.Y. Zhao, A.D. Joseph, R.H. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *NOSSDAV*, June 2001.

The Design of the OpenBSD Cryptographic Framework

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Jason L. Wright
OpenBSD Project
jason@openbsd.org

Theo de Raadt
OpenBSD Project
deraadt@openbsd.org

Abstract

Cryptographic transformations are a fundamental building block in many security applications and protocols. To improve performance, several vendors market hardware accelerator cards. However, until now no operating system provided a mechanism that allowed both *uniform* and *efficient* use of this new type of resource.

We present the OpenBSD Cryptographic Framework (OCF), a service virtualization layer implemented inside the kernel, that provides uniform access to accelerator functionality by hiding card-specific details behind a carefully-designed API. We evaluate the impact of the OCF in a variety of benchmarks, measuring overall system performance, application throughput and latency, and aggregate throughput when multiple applications make use of it.

We conclude that the OCF is extremely efficient in utilizing cryptographic accelerator functionality, attaining 95% of the theoretical peak device performance, and over 800 Mbit/sec aggregate throughput using 3DES. We believe that this validates our decision to opt for ease of use by applications and kernel components through a uniform API, and for seamless support for new accelerators. Furthermore, our evaluation points to several bottlenecks in system and operating system design: data copying between user and kernel modes, PCI bus signaling inefficiency, protocols that use small data units, and single-threaded applications. We offer several suggestions for improvements and directions for future work.

1 Introduction

Today's computing systems are used for applications such as electronic commerce, tele-collaboration of various types, and evolving peer-to-peer systems, often containing sensitive information. Security in these systems depends on several mechanisms that utilize cryptographic primitives as a basic building block. Such cryptographic primitives can be very complex [2] because

the design of these systems is intended to impede simple, brute-force, computational attacks. This complexity drives the belief that strong security is *fundamentally* inimical to good performance.

This belief has led to the common predilection to avoid cryptography in favor of performance [22]. However, the foundation for this belief is often software implementation [8] of algorithms intended for efficient hardware implementation. To address this issue, vendors have been marketing hardware cryptographic accelerators that implement several cryptographic algorithms used by security protocols and applications. However, modern operating systems lack the necessary support to provide *efficient* access to such functionality to applications and the operating system itself through a *uniform* API that abstracts away device details. As a result, accelerators are often used directly through libraries linked with applications, typically requiring device-specific knowledge by the applications, and preventing the operating system itself from easily utilizing such hardware.

We present the OpenBSD Cryptographic Framework (OCF), a service virtualization layer implemented inside the kernel, that provides uniform access to accelerator functionality by hiding device-specific details behind a carefully-designed API. The abstraction introduced allows us to easily support new hardware accelerators and enable applications to use any such accelerator without device-specific knowledge. Furthermore, this intermediate layer does not unduly impact performance, as is common when such abstractions are introduced. The OCF has been in use with OpenBSD [5] for over three years and has proven stable and efficient in practice. It offers features such as load-balancing across multiple accelerators, session migration, and algorithm chaining. We describe the changes we made to the OpenBSD kernel and applications to take advantage of the OCF. In previous work [18] we presented a preliminary analysis of the impact of hardware acceleration on network security protocols, without describing the OCF itself in any detail. Here, we evaluate the impact of the OCF in a variety of micro-benchmarks, measuring overall system perfor-

mance, application throughput and latency, and aggregate throughput when multiple applications use the OCF.

Our evaluation shows that, despite its addition in the system as a device/service virtualization layer, the OCF is extremely efficient in utilizing cryptographic accelerator functionality, attaining 95% of the theoretical peak device performance. In another configuration, we were able to achieve a 3DES aggregate throughput of over 800 Mbps, by employing a multi-threaded application and load-balancing across multiple accelerators. Furthermore, use of hardware accelerators can remove contention for the CPU and thus improve overall system responsiveness and performance for unrelated tasks. Our evaluation allowed us to determine that the limiting factor for high-performance cryptography in modern systems is data copying and the PCI bus. Furthermore, small data-buffers should be processed in software if possible, freeing hardware accelerators to handle larger requests that better amortize the system and PCI transaction costs. On the other hand, multi-threading results in increased utilization of the OCF, improving *aggregate* throughput. We make recommendations for future directions in architectural placement of cryptographic functionality, operating system provisions, and application design, and discuss several improvements and promising directions for future work.

The framework has been in use with IPsec since OpenBSD 2.8, although it continues to evolve in response to new requirements. Public-key support and the */dev/crypto* interface were introduced in a later version. The OCF has also been ported to FreeBSD and NetBSD, and we are working on Windows and Linux versions.

Paper Organization Section 2 discusses related work. Section 3 describes the OCF's design and implementation, while Section 4 discusses its use by various subsystems and applications. In Section 5, we evaluate the framework's performance, and discuss some of the results and potential improvements and future work in Section 6. Section 7 concludes the paper.

2 Related Work

As interest in security is currently in an upswing, recent work has been examining the overall performance impact of security technologies in real systems. Work by Coarfa, *et al.* [4] has focused on the impact of hardware accelerators in the context of TLS web servers using a trace-based methodology, and concludes that there is some opportunity for acceleration, but given the

choice one might prefer a second processor as it also assists with the substantial (and perhaps dominant) non-cryptographic overheads. [18] provides some basic performance characterizations of IPsec as well as other network security protocols, and the impact acceleration has on throughput. The authors conclude that the relative cost of high-grade cryptography is low enough that it should be the default configuration.

There has been a considerable amount of work on the enhancement of system performance through the addition of cryptographic hardware [2]. This early work was characterized by its focus on the hardware accelerator rather than its implications for overall system performance. [24] began examining cryptographic subsystem issues in the context of securing high-speed networks, and observed that the bus-attached cards would be limited by bus-sharing with a network adapter on systems with a single I/O bus. A second issue pointed out in that time frame [20] was the cost of system calls, and a third [21, 23, 7, 11] the cost of buffer copying. These issues are still with us, and continue to require aggressive design to reduce their impacts.

[25] describes an API to cryptographic functions, the main purpose of which is to separate cryptographic libraries from applications, thus allowing independent development. Our service API is similar at a high level, although several differences were dictated by the need to support actual hardware accelerators and allow it to be used efficiently by protocols such as IPsec and SSL, as we discuss in Section 3. Other work includes the Microsoft CryptoAPI [17], GSS-API [16] and IDUP-GSS-API [1], PKCS #11 [14], SSAPI [26], and the CDSA [19]. These are primarily intended for use by applications that also require authentication, authorization, key management and other higher level security services. Our work focuses on low-level cryptographic operations, providing a simple abstraction layer that does not significantly impact performance, compared to a device-specific approach.

[10] describes an open-source cryptographic coprocessor, focusing on protecting keys and other sensitive information from tampering by unauthorized applications. The author extends the *cryptlib* library to communicate with the co-processor. While he discusses several options for hardware acceleration and identifies some potential performance bottlenecks, it is mostly a qualitative analysis. That work is extended in [9], which presents a comprehensive cryptographic security architecture, again focusing primarily on preserving the confidentiality of users' (and applications') cryptographic keys. We are interested in a much simpler problem: how to accelerate cryptographic operations in a general-

purpose operating system using hardware available in the market and with minimal modifications to the kernel, libraries, and applications.

NetBSD uses the `dmove` facility, which provides an interface to hardware-assisted data movers. This can be used to copy data from one location in memory to another, clear a region of memory, fill a region of memory with a pattern, and perform simple operations on multiple regions of memory, such as XOR, without intervention by the CPU.

3 The Cryptographic Framework

The OpenBSD cryptographic framework (OCF) is an asynchronous service virtualization layer inside the kernel, that provides uniform access to hardware cryptographic accelerator cards. The OCF implements two APIs for use by other kernel subsystems, one for use by *consumers* (other kernel subsystems) and another for use by *producers* (crypto-card device drivers). The OCF supports two classes of algorithms: symmetric (*e.g.*, DES, AES, MD5) and asymmetric (*e.g.*, RSA).

Symmetric-algorithm (*e.g.*, DES, AES, MD5, compression algorithms, *etc.*) operations are built around the concept of the *session*, since such algorithms are typically used for bulk-data processing, and we wanted to take advantage of the session-caching features available in many accelerators. Asymmetric algorithms are implemented as individual operations: no session caching is performed. Session creation and teardown are synchronous operations.

The producer API allows a driver to register with the OCF the various algorithms it supports and any other device characteristics (*e.g.*, support for algorithm chaining, built-in random number generation, *etc.*). The device driver also registers four callback functions that the OCF uses to initialize, use, and teardown symmetric-algorithm sessions, and to issue asymmetric-algorithm requests. The drivers can also selectively de-register algorithms, or remove themselves from the OCF (*e.g.*, a PCMCIA card that is ejected). Any sessions using the defunct driver (or algorithm) are migrated to other cards on-demand (*i.e.*, as the next request for that session arrives). Registration and de-registration can occur at any time; typical device drivers do so at system initialization time. Drivers call the `crypto_done()` and `crypto_kdone()` routines to notify the OCF of completed symmetric and asymmetric requests, respectively. A brief description of the API is given in Appendix A.

In addition to any hardware drivers, a software-crypto pseudo-driver registers a number of symmetric-key algorithms when the system boots. The pseudo-driver acts as a last-resort provider of crypto services; any suitable hardware accelerator will be treated preferably. However, the kernel does not implement asymmetric algorithms in software, for performance reasons; we shall see in Section 4.2 how we handle these. Using a generic API for crypto drivers allows us to easily add support for new cards. We briefly discuss these drivers in Section 3.1.

To use the OCF, consumers first create a session with the OCF using `crypto_newsession()`, specifying the algorithm(s) to use, mode of operation (*e.g.*, CBC, HMAC, *etc.*), cryptographic keys, initialization vectors, and number of rounds (for variable-round algorithms). The OCF supports algorithm-chaining, *i.e.*, performing encryption and integrity-protection in one operation. Such combined operations are used by practically all data-transfer security protocols. At session-creation time, the OCF determines which card to use based on its capabilities and creates a session by calling its `newsession` method, provided at device-registration time. When the session is not needed, `crypto_freesession()` frees any allocated resources.

For the actual encryption/decryption, consumers use `crypto_dispatch()`. The arguments to this include the data to be processed, a copy of the parameters used to initialize the session, consumer-provided opaque data, and a callback function. The data can be provided in the form of *mbufs* (linked lists of data buffers, used by the network subsystem to store packets) or as a collection of potentially non-contiguous memory blocks, called *uio*. The case of a single contiguous data buffer is handled as a *uio*. Although *mbufs* are also a special case of *uio*, we added special support to allow for some processing optimizations when using software cryptography. Furthermore, the issuer of a request can specify whether encryption should be done in-place, or the encrypted data must be returned on a separate buffer. Various offsets indicate where to start and end the encryption, where to place the message authentication code (MAC), and where to find the initialization vector (if already present on the buffer) or where to write it on the output buffer.

The request is queued and `crypto_dispatch()` immediately returns to the consumer. The `crypto` kernel thread is periodically invoked by the scheduler and dispatches all pending requests to the appropriate producers. It also handles all completed requests, by calling the specified callback functions. It then returns to sleep, waiting for more requests. As a result of the OpenBSD kernel architecture (common in most non-SMP kernels), the thread

is not preemptable by user processes, although hardware interrupts are still handled. Currently, the thread must operate at a high priority to avoid synchronization problems. When using the software pseudo-driver, this can cause significant latency in application scheduling and in low-priority kernel operations, although the same problem manifested before the migration to OCF, when encryption was done in-band with IPsec packet processing.

Once the request is processed, the crypto thread calls the consumer-supplied callback routine. If an error has occurred, the callback is responsible for any corrective action. Session migration is implemented by re-creating the session using the initial parameters to *crypto_newsession()*, which accompany all requests as we already mentioned. The error **EAGAIN** is indicated to the callback routine, which re-issues the request after recording the new session number to be used so that subsequent requests are correctly routed. Including the initialization data in each request also allows us to easily integrate cards that do not support the concept of session: the driver simply passes all necessary information (data, algorithm descriptions, and keys) to the card with each request. The opaque data are simply passed back to the consumer unmodified by the OCF; they are used to maintain any additional information for the consumer that is relevant to the request. We shall see an example in Section 4.1.

Asymmetric operations are handled similarly, albeit without support for the concept of session. The parameters in this case include an array of parameters, containing the algorithm-specific big-integers.

When multiple producers implement the same algorithms, the OCF can load-balance sessions across them. This is currently implemented by simply keeping track of the number of sessions active on each producer. At session setup, the OCF picks the producer with the smallest number of active sessions. The software pseudo-driver is currently never used in load-balancing. We evaluate the effectiveness of this simple scheme in Section 5.4. We discuss possible future improvements in Section 6.4.

3.1 Device Drivers

The drivers for the various crypto devices must be able to cope with a wide variety of hardware design decisions (and bugs) made by the manufacturers. These drivers register the algorithms supported by the device and export the appropriate callback functions to the OCF.

The *hifn* driver supports the Hifn 7751, 7811, and 7951 chips and contains around 3,000 lines of code and def-

initions. The driver supports the symmetric operations and hashes available on all these chips. Additionally, it supports the random-number generators available on the 7811 and 7951, but does not support the public key unit on the 7951; the latter was clearly designed for SSL server implementations, as it requires a large amount of CPU-intensive initialization which can be precomputed and used repeatedly on a server but not a client. All these chips support copying-through header and trailer data to the destination buffer, and include full support for scatter-gather I/O. Unfortunately, there is no easy way to coalesce interrupts on this chip, which generates one interrupt per operation, resulting in considerable system overhead. Another important detail is that all of the Hifn symmetric crypto chips poll their descriptor rings in main memory for data to process.

The *nofn* driver supports the Hifn 7814, 7851, and 7854 chips (also known as HIPPI packet processors). Currently, there is no support for the symmetric unit on these chips. Fitting these into the current framework is not currently done because they are designed to replace almost all of the IPsec processing (IV generation, MAC checking, replay window handling, *etc.*). In the future, we intend to add support for the IPsec unit by adding a combined-class algorithm and checking for this in IPsec. On the other hand, the public-key unit is almost exactly the same as the Hifn 6500 described below.

The *lofn* driver supports the Hifn 6500 chip, which contains a public-key unit and a random-number generator. This chip is essentially a simple big-number arithmetic logic unit (*i.e.*, it is an ALU capable of performing operations on 1024-bit registers). Unlike all of the other chips, the 6500 is not a bus-master (*i.e.*, has no support for DMA); instead, registers exist within its PCI memory-mapped address space. Because of the expense of modular exponentiations, the somewhat higher overhead of writes to these I/O addresses is still small compared to doing the exponentiation in software.

The *ubsec* driver, which supports the Broadcom 5801, 5802, 5805, 5820, 5821, and 5822 chips, consists of slightly less than 3,000 lines of code and definitions. The symmetric-crypto units on all of the chips are very similar, but the 580x series and 582x series require different formatting for the big numbers on the asymmetric unit. These chips support interrupt coalescing by chaining several commands together, and scatter-gather I/O. Unlike Hifn, these chips do not poll main memory.

We have a variety of other device drivers in various stages of completion. We are aware of other and more modern products from a variety of vendors, but many of them are hesitant to give us the information we need.

4 Use of the OCF in OpenBSD

In this section, we discuss how we extended parts of OpenBSD to make use of the OCF services.

4.1 IPsec

The IP Security Architecture [12], as specified by the Internet Engineering Task Force (IETF), is comprised of a set of protocols that provide data integrity, confidentiality, replay protection, and authentication at the network layer. At the lowest level of the IPsec architecture reside the data encryption/authentication protocols, AH and ESP. These are the “wire protocols,” used for encapsulating the IP packets to be protected. They simply provide a format for the encapsulation; the details of the bit layout are not particularly important for the purposes of this paper. Outgoing packets are authenticated, encrypted, and encapsulated just before being transmitted, and incoming packets are decapsulated, verified, and decrypted immediately upon receipt. These protocols are typically implemented inside the kernel, for performance and security reasons.

IPsec was the first consumer of the OCF services. The original implementation of the OpenBSD IPsec was described in [13]. Here, we give a brief overview and then describe the modifications we had to make to it to allow use of the OCF.

In the OpenBSD kernel, IPsec is implemented as a pair of protocols sitting on top of IP. Thus, incoming IPsec packets destined to the local host are processed by the appropriate IPsec protocol through the protocol switch structure used for all protocols (e.g., TCP and UDP). The selection of the appropriate protocol is based on the protocol number in the IP header. The SA needed to process the packet is found in an in-kernel database using information retrieved from the packet itself. Once the packet has been correctly processed (decrypted, integrity-validated, etc.), it is re-queued for further processing by the IP module, accompanied by additional information (such as the fact that it was received under a specific SA) for use by higher-level protocols and the socket layer.

Outgoing packets require somewhat different processing. When a packet is handed to the IP module for transmission (in `ip_output()`), a lookup is made in the Security Policy Database (SPD) to determine whether that packet needs to be processed by IPsec. The decision is made based on the source/destination addresses, transport protocol, and port numbers. If IPsec processing is needed, the lookup will also specify what type of

SA(s) to use for IPsec processing of the packet. If no suitable SA exists, the key-management daemon is notified to acquire one. Otherwise, the packet is processed by IPsec and passed to `ip_output()` again for transmission. The packet also carries an indication as to what IPsec processing has already occurred to it, to avoid processing loops.

In the original IPsec implementation, all cryptographic operations were done in-band with packet processing. This meant that a lot of time was spent performing symmetric-key encryption in the kernel. To make use of the OCF, we split the input and output processing paths. For example, let us consider the case where `ip_output()` determines (by consulting the SPD) that a packet must be IPsec-protected. It then calls `ipsec_process_packet()`, which handles all IPsec outbound-packet processing. After handling encapsulation issues, this routine calls the appropriate “wire protocol” output routine. In the ESP protocol processing, the original `esp_output()` routine was broken up in `esp_output()` and `esp_output_cb()`. `esp_output()` does all the data marshaling and ESP header manipulation, constructs a crypto request, passes it to the OCF and simply returns. Execution returns to `ip_output()` with an indication that the operation was successful.

Once the OCF processes the request, it calls `esp_output_cb()`, a pointer to which is included in the request itself. The callback routine completes the ESP protocol processing by checking for any errors in the crypto processing (re-queuing the request if the OCF indicated so), and calls `ipsec_process_done()`, the second part of the original `ipsec_process_packet()` routine. This routine completes IPsec book-keeping, and calls `ip_output()` with the new packet. `ip_output()` will then perform a new SPD lookup (making sure no IPsec loops occur, by examining the list of SAs that have been already applied to the packet). If necessary, the output processing cycle will occur again. Eventually, `ip_output()` will pass the packet to a network driver for actual transmission.

The cases for output AH and IPcomp processing are similar. Input processing is also similar: `ipsec_common_input()` is called by the network scheduler for all IPsec packets received. It locates the appropriate SA in the kernel SA database and calls `esp_input()`. Similar to the output case, `esp_input()` validates the ESP header fields, constructs a crypto request, passes it to the OCF and returns. Once the request is processed, the OCF will call `esp_input_cb()`, which will verify the packet integrity (by comparing the value on the packet with that computed by the accelerator), remove the ESP header, and pass the packet to `ipsec_common_input_cb()`.

This routine performs further sanity and security checks on the decrypted packet, and re-queues it for further processing by the IP layer. AH and IPcomp input processing is similar, as is the case of IPsec over IPv6.

Input ESP and AH processing offer one example of use of the opaque data passed with each crypto request, discussed in Section 3. All the cryptographic accelerators that support message authentication (MAC) algorithms only offer a “forward-compute” mode. That is, the card can only compute the MAC on the packet, and it is up to the operating system to verify its validity by comparing it with the received value. Thus, we use the opaque data to store the MAC value from the packet and instruct the OCF to write the new MAC value in the appropriate location in the packet — the operation is exactly the same as the output case. In the callbacks, we simply do a byte-wise comparison of the computed value (stored on the packet) and the received value (stored as opaque data in the request itself).

While the code was not very complicated, there were several minor headaches as a result of this asynchronous processing model. For example, one problem was communicating MTU information through arbitrarily-many IPsec SAs to the TCP layer, so as to correctly fragment application data and avoid fragmentation at the IP layer. We could not simply update the appropriate data structures with the correct MTU value after the packet had been encapsulated once, since we could not “peek” inside the encryption. Fortunately, we keep a record of which SAs have been applied to a packet during input and output processing. Thus, on receipt of the appropriate ICMP message, or when the IP layer indicates that the packet is too large to be transmitted without fragmentation, the list of SAs is traversed and each SA is updated with the correct MTU value based on its position in the SA chain (*i.e.*, the first SA on output will advertise a smaller MTU than the last one, the difference being the ESP headers and encryption padding). The next packet that tries to traverse the chain will encounter a correct MTU value.

4.2 /dev/crypto

Building on our experience with the IPsec implementation, we turn our attention to exporting the OCF services to user-level applications. A `/dev/crypto` device driver exists which abstracts all the OCF functionality and provides a command set that can be used by OpenSSL (or other software that uses `/dev/crypto` directly). This interface is based on `ioctl()` calls and is thus fully synchronous (*i.e.*, applications can only have one request pending) — in the future, we intend to al-

low processes to issue multiple requests. Both symmetric and asymmetric operations are permitted using this framework; we will first describe the symmetric component.

Similar to the underlying OCF, this uses a session-based model, since the general case assumes that keys will be reused for a sequence of operations. After opening the `/dev/crypto` device and gaining a file descriptor `fd`, the caller requests that a new session be created with `CIOCGSESSION` for a certain cryptographic operation, and specifies all related parameters (*e.g.*, keys). Similar to the OCF, a single session supports both a cipher and a MAC, as we are simply exporting the same functionality available to the kernel. `CIOCGSESSION` returns a session identifier that can then be reused repeatedly for subsequent operations. When the session is no longer needed, it can be revoked using `CIOCFSESSION`. Many sessions can be requested against a single file descriptor `fd`; all sessions follow a particular `fd` through `fork()` and `exec()` calls, and are not otherwise visible to other processes. Obviously, the last `close()` on `fd` destroys all the sessions.

If the request cannot be satisfied using hardware accelerators, the kernel will return an error of `EINVAL`, so the caller can fall back to a software implementation. We considered adding an `ioctl()` that describes the abilities of the available hardware, allowing an application to determine if the needed algorithm is supported by looking at a list. However, numerous other variables exist (key sizes, block sizes, alignment) which might be difficult to describe. For the time being, we have punted on this issue. However, when first called, the OpenSSL engine will enumerate all OCF-supported algorithms. It does so by issuing a `CIOCGSESSION` request for each algorithm it supports in software, and caches the result. If an algorithm is not provided by the OCF, the library will use its software implementation (in reality, the kernel will admit that it supports cryptographic algorithms that it implements in software, and OpenSSL will make use of them as if they were implemented by hardware, unless a `sysctl` variable is set to prohibit this, which is the default setting).

Once a session is established, blocks can be encrypted or decrypted using the `CIOCCRYPT ioctl()`. Each time this is used, the caller can specify a new IV or MAC information that they wish to fold into the operation. Input and output buffers are specified via separate pointers, but they can point to the same buffer for in-place encryption. Naturally, the data size provided by the caller must be rounded to the default block size of the algorithm being used. A data size limit of 262,140 bytes exists at the moment, to hide a similar limit found in some chipsets.

In the future, we may support larger blocks by splitting operations into smaller chunks.

The user-land data blocks are copied into memory allocated inside the kernel address space. This data is formatted into *uio* blocks as mentioned in Section 3. The OCF is then called to perform the operation using the initialization information stored in the application's `/dev/crypto` session. If the operation is successful, the results are copied back to the application buffers. Obviously, the cost of these two copies is higher for larger block sizes, as we shall see in Section 5.4. In the future, we hope to use page flipping for larger blocks when the kernel memory subsystem supports this.

For asymmetric operations, no session is required. The `CIOCKEY ioctl()` is used in an atomic fashion for each individual operation. Five operations are provided, with `CRK_MOD_EXP` being the most important. Support for the others, `CRK_MOD_EXP_CRT`, `CRK_DSA_SIGN`, `CRK_DSA_VERIFY`, and `CRK_DH_COMPUTE_KEY` has not yet been completed. Each of these has an operation-specific number of input and output parameters, which are always a packed byte array of big integers. The particular format we chose for these parameters makes it easy to interface to OpenSSL “bignums,” and to most of the early hardware we had access to.

Presently, OpenBSD lacks cloning devices. Therefore a cumbersome procedure for opening `/dev/crypto` must be followed. After the initial `open()` call, the caller must use `ioctl()` to retrieve a file descriptor (*fd*) to use, then perform all operations against this replacement *fd*. This replacement *fd* is a unique per-process descriptor, while the initially-opened one would naturally be shared between all callers. Without such semantics, the `fork()` and `_exit()` system calls do not exhibit the expected semantics with respect to file-descriptor inheritance and closing. Just as bad, we would end up with all processes able to see and use each other's keys. When cloning devices are implemented in OpenBSD, we will change the user-level code (mostly OpenSSL) to no longer use this complicated procedure, but the kernel will retain it for backward compatibility. While writing this code, we ran into numerous strange and difficult resource-management issues for session teardown.

It should also be noted that applications using `/dev/crypto` must ensure they use `ioctl()` with the `F_SETFD` command on the crypto descriptor to ensure that the “close-on-exec” flag is set. Otherwise, child processes will inherit unwanted descriptors, which is both a security and a resource-exhaustion concern.

4.2.1 OpenSSL Enhancements

In the past, programmers using OpenSSL (or its predecessor, SSLeay) directly called the generic crypto routines as they existed for each algorithm. More recently, programmers have been encouraged to use the *EVP* layer for dealing with symmetric algorithms. This provides a session-based model much like the `/dev/crypto` layer described in the previous section. Applications like OpenSSH, *mod_ssl* (the Apache SSL module we use), and *sendmail* have matured to use these interfaces.

Newer OpenSSL code-bases contain an “engine” component. This allows asymmetric algorithms to be directed to a hardware driver; a number of stub functions are provided which typically interface with vendor-specific shared libraries to actually do the operation on the vendor's accelerator. Many of these subsystems interact badly and do not consider the effects of *chroot()* or other strange Unix behaviors, resulting in weak security models. Since we run Apache in a *chroot()*'ed environment in which there exists no `/dev/crypto` device, we modified it to perform all necessary initializations prior to being sandboxed. We wrote our own engine modules that interacts directly with `/dev/crypto`, without any of these surprises. Symmetric operations from the *EVP* layer are directly mapped into OCF requests. One major weakness is that the *EVP* layer has no concept of bundling algorithms. Thus, protocols that use encryption and MAC on a message, such as TLS and SSH version 2, sequentially issue two separate requests to `/dev/crypto` through the *EVP* layer, resulting in unnecessary context switches, data copying, and DMA transactions. Thus, the *EVP* layer currently does not pass MAC operations to the OCF.

5 Performance Evaluation

In this section, we analyze the performance of the cryptographic framework. We have ran a series of micro-benchmarks that allowed us to determine the limits of the framework and potential directions for improvement. We use the OCF for simple cryptographic tasks, comparing different cryptographic accelerators with the case of pure-software encryption, and provide a cost breakdown. We also attempt to quantify the benefits to be had by the system at large, when off-loading cryptographic operations to hardware accelerators. Finally, we evaluate the load-balancing feature of OCF, by simultaneously using multiple accelerators on the same machine.

5.1 Testbed

For our tests, we use two identical machines. The machines have 1.4 Ghz Pentium III processors on Tyan Thunder HESl-T motherboards. These motherboards have three independent PCI busses: 32bit/33Mhz/5V, 64bit/66Mhz/5V, and 64bit/66Mhz/3.3V. The boards use 512MB of 133Mhz registered SDRAM and are based on the ServerWorks HESL chipset. We placed the crypto card being tested either on the 64bit/66mhz/3.3V bus or the 32bit/33Mhz/5V bus, as appropriate for the card. The crypto cards we used are:

- Broadcom 5805 reference design board (32bit).
- Broadcom 5820 reference design board (64bit).
- GTGI XL-Crypt (based on the Hifn 7811 chip) (32bit).
- NETSEC 7751 (based on the Hifn 7751 chip) (32bit).
- Hifn 6500 reference design board (32bit).
- Hifn 7814 reference design board (64bit).

The Hifn data-sheet gives a peak performance for the 7751 chip of 62 Mbps for encryption and 110 Mbps decryption, when using IPsec with 3DES/SHA1/LZS (LZS is a data-compression algorithm). When the 3DES engine alone is used, both encryption and decryption throughput are 83 Mbps. Broadcom's web site places the peak performance of the 5820 chip at 310 Mbps of 3DES-SHA1, when used in IPsec. Furthermore, they claim 800 1024-bit RSA signature computations per second.

5.2 OCF Throughput

To determine the raw performance of OCF, we use a single-threaded program that repeatedly encrypts and decrypts a fixed amount of data with various symmetric-key algorithms, using the `/dev/crypto` interface. We run the test against all the hardware accelerators listed in the previous section, as well as using the kernel-resident software implementation of the algorithms. We vary the amount of data to be processed per request across experiments. To measure the overhead of OCF without the cryptographic algorithms, we added to the kernel a *null* algorithm that simply returns the data to the caller without performing any processing. The results can be seen in Figure 1.

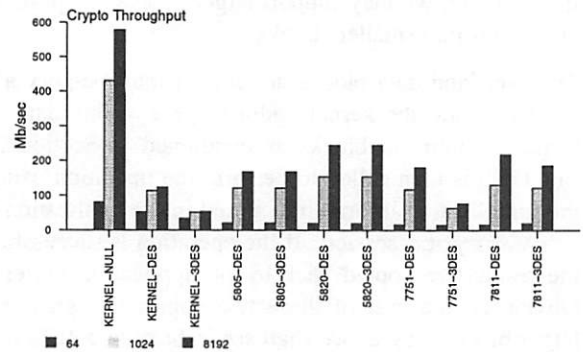


Figure 1: Crypto-hardware performance. The *KERNEL-NULL* bar indicates use of the *null* encryption algorithm. The *KERNEL-DES* and *KERNEL-3DES* bars indicate use of the software DES and 3DES implementations in the kernel. The remaining bars indicate use of the various hardware accelerators. The vertical axis unit is Mbits/second.

We can make several observations on this graph. First, even when no actual crypto is done, the ceiling of the throughput is surprisingly low for small-size operations (64 bytes). In this case, the measured cost consists of the overhead of system call invocation, argument validation, and crypto-thread scheduling. As larger buffers are passed to the kernel, the throughput increases dramatically, despite the increasing cost of memory-copying larger buffers in and out of the kernel. When we use 1024-byte buffers, performance in the no-encryption case jumps to 420 Mbps; for 8192-byte buffers, the framework peaks at about 600 Mbps.

Notice however that this peak corresponds to a single process issuing crypto requests. This process is blocked after each request, the scheduler context-switches to the crypto thread (which was blocked waiting for requests), the *null* algorithm executes and the completed request is passed back to the `/dev/crypto` driver, which wakes up the blocked user-level process. If many processes are issuing requests, the crypto thread's request queue will contain multiple requests. When we run multiple processes, each will queue a request (and be blocked by `/dev/crypto`); the crypto thread will process all these requests in a flurry of activity, and cause all processes to wake up in synchrony. The crypto thread will then go back to sleep, while each of the processes will issue another request. This cycle repeats for the duration of the experiment. As a result, more processes using the OCF result in increased aggregate throughput, simultaneously increasing the average processing latency.

These buffer sizes are close to the typical sizes of re-

quests issued by some of the most-commonly used applications:

- *SSH* keyboard input results in many small requests (so we are close to the 64-byte case); responses from the server are larger, but not considerably so. When X forwarding is used, we can occasionally get larger buffers.
- *SCP/SFTP* issue larger requests; OpenSSH, a popular implementation, uses requests of 4 KB.
- *SSL/TLS* also issue large requests. The maximum size of an SSL record is 16 KB, but can be less if (optional) compression is used.
- *IPsec* processes packets at the network layer. Such traffic is trimodal [3]: about 40% of packets are 40–60 bytes (the vast majority of these being TCP acknowledgments), with the remainder split between 576 bytes (TCP MSS when no Path MTU Discovery is used) and 1460 bytes (when Path MTU Discovery is used).

When we use real cryptographic algorithms, we notice that the performance of DES done in software is close to that of no encryption for small packet sizes; even 3DES performance is just half of the no-encryption case. If we use larger buffer sizes, the performance of software crypto done in the kernel (the *KERNEL-** labeled bars) degrades rapidly. When we use hardware accelerators, we notice two different trends. For small buffers, the performance degrades with respect to the software case. This indicates that the additive costs of system call invocation, OCF processing, and the 2 PCI transactions (to/from the crypto cards) dominate the cost of doing crypto. However, as we move to larger buffer sizes, performance quickly improves as these overheads are amortized over larger buffers, despite the fact that more data has to be copied in and out of the kernel and over the PCI bus. Thus, to improve the performance of the system when applications issue large numbers of small requests, either request-batching should be done, a faster processor should be used, or the number of user/kernel crossings should be minimized. When larger buffers are being processed, it pays off to use some cryptographic accelerators, although not all such cards are equal in terms of performance.

Notice that the performance of DES and 3DES is the same in each of the 5805 and 5820 cards; these cards really implement only 3DES in Encrypt-Decrypt-Encrypt (EDE) mode, and emulate DES by loading the same key in one of the Encrypt and the Decrypt engines (effectively canceling each other out). In contrast, the 7751

seems to implement two separate crypto engines for DES and 3DES, or uses a shortcut in its 3DES engine. The 7811 seems to implement different engines as well, but the performance difference between the two is not as pronounced.

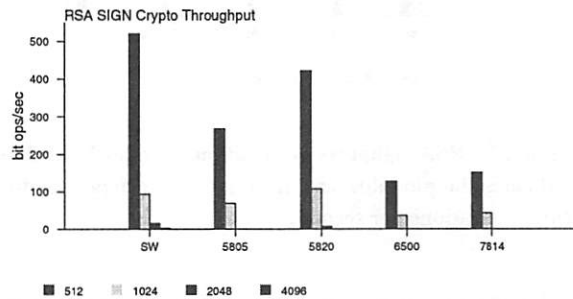


Figure 2: RSA signature generation. The horizontal axis indicates the modulus size, in bits. The vertical axis indicates number of operations per second.

Similarly, we measure the performance of OCF for public-key operations. In this case, there are no kernel-resident software public-key algorithms. We count the number of RSA signature generations and verifications per second, for different accelerators and key sizes (512 to 4096 bits, as supported by the each cards). The results are shown in Figures 2 and 3.

The Hifn 6500 and 7814 are geared more towards slower, embedded applications, so the fact that their performance is considerably worse than software is not surprising. The number of verifications is much larger than the number of signature generations in unit time. This is because, as with most crypto libraries, OpenSSL opts for small values for the public part of the RSA key (typically, $2^{16} + 1$) and correspondingly large values for the private key. This causes the public-key operations (encryption and verification) to be much faster than the private-key operations, even though they are in principle the same operation (modular exponentiation).

Another interesting observation is that the RSA sign throughput is higher in the software case (see Figure 2). This happens because the CPU on the crypto-card is slower than the host CPU and optimized for bit operations, which is as useful for public key cryptography. So the “anomaly” in Figure 2 is actually expected. However, as we mentioned in Section 5.1, Broadcom claims that the 5820 can perform 800 RSA signature operations per second with 1024-bit keys. In our case, we only see slightly over 100. There are two explanations for this. First, we are under-utilizing the 5820: there is only one thread issuing RSA sign operations, which is blocked waiting termination of each request. Once the card com-

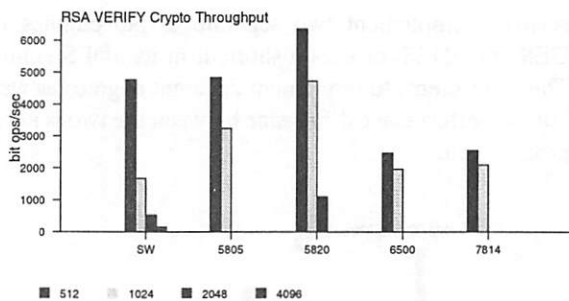


Figure 3: RSA signature verification. The horizontal axis indicates the modulus size, in bits. The vertical axis indicates operations per second.

puts the signature, it has to wait for the crypto framework to wake up the blocked process, then the scheduler to context-switch to it, the process to issue an *ioctl()* call to get the results, and then another *ioctl()* call to issue the next request, which is placed on the crypto thread's queue. Finally, the scheduler has to context-switch to the crypto thread. During all this time, the accelerator is idle, since there is no other process using it. The second reason for the higher vendor-stated performance is that the tests they performed used the CRT parameters for the RSA operations, which make RSA processing considerably faster. However, for implementation reasons, our OpenSSL engine does not use CRT parameters yet.

5.3 System-wide Effects

To determine the system-wide benefits of offloading cryptographic processing, we run multiple threads (up to 24) of the `openssl speed` benchmark with various algorithms, while at the same time we run a simple CPU-intensive job. The CPU "hog" process consists of a small program that performs 2^{32} function calls, each function call performing an integer-multiply operation. The elapsed time for the CPU hog process was recorded for each (*algorithm, number of threads*) tuple. As we see in Figure 4, the crypto accelerators very effectively eliminate contention for the otherwise-shared resource, the CPU, whether the crypto performed is symmetric (DES, 3DES) or asymmetric (DSA with 1024-bit keys). The execution time for the hog process remains constant, regardless of the number of threads of execution.

5.4 Load Balancing

Finally, we wish to determine how well the OCF can load-balance crypto requests when multiple accelerators

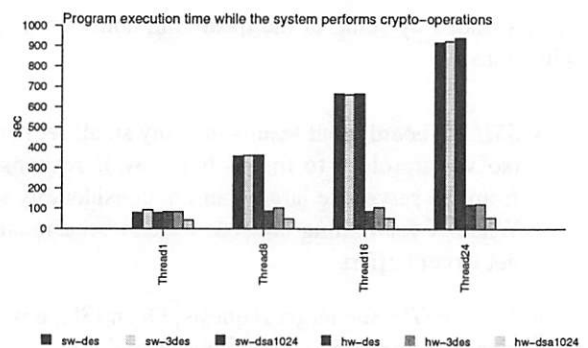


Figure 4: Program execution time while multiple threads perform crypto-operations in parallel. The bars show the elapsed time in seconds for executing the CPU-bound process for different algorithms and numbers of threads.

are available, and the aggregate throughput that can be achieved in that scenario. We use a custom-made card by Avaya that contains four Hifn 7751 chips that can be used as different devices through a PCI bridge resident on the card. We use multiple threads that issue encryption requests for 3DES, and vary the buffer size across different runs. The results are shown in Table 1. As we can see, performance peaks in the case of 32 threads and 16 KB buffers at 320 Mbps, which is over 96% of the maximum rated throughput of four Hifn 7751 chips. The card was installed on the 64bit/66Mhz PCI bus, but because the chip is a 32bit/33Mhz device, the maximum bus transfer rate is 1.056 Gbps. At our peak rate, we use over 640 Mbps of the bus: 320 Mbps for data in each direction (to and from the card), plus the transfer initialization commands and descriptor ring probing, *etc.*, thus utilizing over 60% of the PCI bus. Notice that because the card uses a PCI bridge, a 2-cycle latency is added on each PCI transaction.

The card was installed on the 64bit/66Mhz bus because the system's 32bit/33Mhz bus exhibited surprisingly bad performance, probably because many other system components are found on that bus and likely cause contention: since the machine is operating as it normally would while this test is being run, the scheduler is active, and two clock interrupts are being received at 100 and 128 Hz respectively. Other devices are also generating their own interrupts.

Another possible cause is an artifact of the i386 *spl* protection method: a regular *spl* subsystem disables the interrupts from a certain class of devices at the invocation of an *splX()* call. For instance, calling *splbio()* blocks reception of interrupts from all devices which are in the "bio" class of devices. On the i386, the registers used

Number of threads	16 bytes	64 bytes	256 bytes	1024 bytes	8192 bytes	16384 bytes
1	3.06 Mbps	11.45 Mbps	33.15 Mbps	59.49 Mbps	79.19 Mbps	80.75 Mbps
2	5.53 Mbps	18.40 Mbps	56.07 Mbps	111.60 Mbps	154.18 Mbps	160.02 Mbps
3	6.44 Mbps	23.25 Mbps	71.31 Mbps	152.28 Mbps	229.60 Mbps	238.24 Mbps
4	6.83 Mbps	25.77 Mbps	80.91 Mbps	182.65 Mbps	292.15 Mbps	299.33 Mbps
32	7.37 Mbps	27.51 Mbps	94.05 Mbps	249.17 Mbps	313.79 Mbps	320.19 Mbps

Table 1: Crypto-request load-balancing using a quad-Hifn 7751 card on a PCI 64bit/66Mhz bus.

Number of threads	16 bytes	64 bytes	256 bytes	1024 bytes	8192 bytes	16384 bytes
1	5.42 Mbps	18.88 Mbps	61.94 Mbps	151.95 Mbps	300.88 Mbps	254.79 Mbps
32	9.91 Mbps	37.01 Mbps	120.71 Mbps	410.27 Mbps	758.85 Mbps	801.81 Mbps

Table 2: Crypto-request load-balancing using four 5820 cards on a PCI 64bit/66Mhz bus.

to do interrupt blocking (found on the programmable interrupt controller, also known as the PIC) are located on the 8Mhz ISA bus, which is what OpenBSD uses for interrupt management (as opposed to the APIC).

Worse yet, some operations on this device require a 1 *usec* delay before taking effect. To partially mitigate this extremely high overhead, the i386 kernel interrupt model instead makes the vectors for blocked interrupt routines point to a single-depth queuing function which does the actual interrupt blocking at the time of reception. When the *spl* is lowered again, the original interrupt handler is called. However, the 8Mhz ISA bus still had to be accessed. This has the effect of further reducing the available bandwidth on the PCI bus. One small-buffer benchmark generated over 62,000 interrupts/sec; we believe that the *spl* optimization is failing under such load.

Using four 5820 cards on a 64bit/66Mhz PCI bus allows us to achieve even higher throughput, as shown in Table 2. We show only the 1 and 32-thread tests; the rest of the measurements followed a similar curve as the quad-7751. Performance peaked at over 800 Mbps of crypto throughput. Using the same analysis as before, we are using in excess of 1.6 Gbps of the fast-PCI bus, which has a throughput of 4.22 Gbps, achieving slightly over 38% utilization of the bus. As we mentioned in Section 5.1, the vendor rates this card at 310 Mbps. Thus, the maximum theoretical attainable rate would be 1.24 Gbps. We achieve 64.5% utilization of the four cards in this case. A rough sampling of CPU utilization during these large block benchmarks on both cards showed around 10,000 interrupts/second, which is substantial for a PC.

Investigating further, we determined that all four 5820 cards were sharing *irq* 11. Thus, it is possible that

the culprit is the *spl* optimization previously mentioned, at least for the small buffer sizes: the *vmstat* utility shows us anything from 50,000 to 60,000 interrupts per second when processing buffers of 16 to 1024 bytes. Furthermore, because of a quirk in the processing of shared *irq* handlers, some cards experience slightly worse interrupt-service latency: shared *irq* handlers are placed in a linked list; if multiple cards raise the interrupt at the same time, the list will be traversed from the beginning for each interrupt raised — and each *irq* handler will poll the corresponding card to determine if the interrupt was issued by it. However, fixing this quirk or moving the cards on different *irq*'s did not significantly improve throughput.

When we use 8192-byte buffers, the interrupt count drops to 12,000, which the system can handle. In each of these cases, the system spends approximately 65% of its time inside the kernel. Most of this cost can be attributed to data copying. However, as we move to larger buffer sizes, we find the system spending 89% of its time in the kernel, and only 1.9% in user applications, for the case of 16 KB buffers. The number of interrupts in this case is only 5,600, which the system can easily handle. The problem here is that there is considerable data copyin/copyout between the kernel and the applications; aggravating the situation, while such data copying is in progress no other thread can execute, causing a “convoy” effect: while the kernel is copying a 16 KB buffer to the application buffer, interrupts arrive that cause more completed requests to be placed on the crypto thread’s “completed” queue. The system will not allow the applications to run again before all completed requests are handled, which cause more data copying. Thus, the queue will almost drain before applications will be able to issue requests again and refill it. We intend to further investigate this phenomenon.

Fundamentally, the data copyin/copyout limitation is inherent in the memory subsystem. We measured its write-bandwidth to be approximately 2.4 Gbps. Using the crypto cards, we are in fact doing 3 memory-write operations for each data buffer: one copyin to the kernel, one DMA from the card to main memory, and one copyout to the application. Notice that data DMA'ed in from the card is not resident in the CPU cache, as all such data is considered "suspect" for caching purposes. In addition, there is an equal amount of memory reads (copyin, DMA in from the card, copyout). Each of those transfers represents an aggregate of 800 Mbps. When we ran the same test with three 5820 cards, performance slightly improved to 841.7 Mbps in the case of 16 KB buffers, achieving over 90% utilization of the three cards. In this case, the memory subsystem is still saturated, but the cards can more easily get a PCI-bus grant and perform the DMA.

6 Discussion

6.1 Cryptography in the Kernel

As we saw in the previous section, the influence of multi-threading on performance is strong, which suggests that busy servers can make better use of hardware cryptography than clients. This supports the observations of Dean, *et al.* [6] that it may make sense to make cryptography a shared network service to achieve the best cost/performance in a secure system. Notice that, within the boundaries of one host (operating system instance), this is precisely what the OCF does. We should also mention that use of a threaded model for applications involves an obvious security vs. implementation complexity trade-off.

Although the performance of individual applications may not improve drastically when using an accelerator, it appears that the *aggregate* performance of a number of applications (as may be the case in a system with many remote login sessions, a busy web server, or a VPN gateway) does improve, as a result of increased utilization. Furthermore, hardware accelerators can give a performance boost to the rest of the system, as was seen in Figure 4. Very simply, they eliminate contention for the CPU, which is a resource shared by all applications and the operating system itself. Thus, while throughput is not drastically improved (and may in fact degrade in certain scenarios) with use of hardware acceleration, overall system utilization improves because the main CPU is left to perform other tasks.

6.2 System Architecture

As we saw in Section 5.4, data copying and the PCI bus quickly become the limiting factor. In practice, the situation is even worse since cryptography is used in conjunction with either network security protocols, in which case the network interface card (NIC) contents for a slice of the PCI bandwidth, or with filesystem encryption, in which case the storage device claims a portion of the bus. This situation suggests that, for maximum performance, cryptographic support must be provided by the individual devices (*e.g.*, NICs, disk controllers, *etc.*). Alternatively, cryptographic support must be located elsewhere in the system architecture (*e.g.*, attached to the main CPU, the system "north bridge" (as the video subsystem is), or the memory subsystem. Any of these approaches, if implemented correctly, will improve application performance by reducing contention for the PCI bus, but at the same time will create new challenges for operating systems that have to support these new devices, such as session migration and fail-over (which the OCF supports by design, as we discussed in Section 3).

Although the OCF does not directly take advantage of NICs that support IPsec-processing offloading, since they are not general-purpose cryptographic accelerators, we have extended the IPsec stack to use them. The cards of this type we are familiar with are 100 Mbps full-duplex Ethernet, and it seems reasonable to assume that they can achieve that performance, given our results with dedicated cryptographic processors. Unfortunately, at the time this paper was written, we did not have enough information to write a device driver that could take advantage of such features. We are also not aware of any commercially-available hard drive controllers that provide built-in encryption services.

6.3 The Effect of Small Requests

The nature of the challenge for operating systems and their support for cryptography is clear. On every measurement, without exception, small-sized operations fare much worse than those performed on large data buffers. In some cases, buffer size influences performance more than the choice between hardware or software cryptography. This suggests that the per-operation overhead is very high, and this is clear from the larger data sizes, which get close to the throughput advertised by the board manufacturer, which we presume is "best-case". In this respect, our findings confirm those of [15]. Since many cryptographic protocols are transactional in nature rather than bulk transfers, these small data operations will be the common case. Energy should be spent on

reducing the overhead of such cases.

As we mentioned in Section 5.2, there are several possible approaches: request-batching, kernel crossing and/or PCI transaction minimization, or simply use of a faster processor. These are more cost-effective solutions than deploying a hardware accelerator. In situations where bulk data transfer is the norm (as may be the case in the various Storage Area Network technologies currently under consideration), cryptographic accelerators can drastically improve performance, especially for the more “expensive” algorithms such as 3DES. Unfortunately, there were no commercially available hardware accelerators for AES supported by OpenBSD, so we cannot compare the software and hardware cases for that algorithm. However, recent attacks against AES make likely the continued use of 3DES in many environments.

6.4 Other Optimizations and Future Work

Smarter load balancing. The load-balancing currently done in OCF, as discussed in Section 3, is very simple. It performs load-balancing of sessions, by keeping a record of the active sessions per producer and selecting the least-loaded one. However, not all sessions are equivalent in terms of processing requirements: an FTP-over-IPsec session will use the OCF more heavily than a telnet-over-IPsec one. Furthermore, the current scheme does not perform load-balancing for public-key operations. Finally, all producers of crypto services are considered equal, in terms of performance. All these issues point to several potential improvements that can be made to the OCF.

For example, drivers can state their peak performance (experimentally measured, using the vendor-provided numbers, or measured at system boot time), and the OCF can keep a record of the number of operations actively pending on each driver. However, this requires sessions to be simultaneously established on all these cards; as these cards have a limited amount of memory for session caching, this approach is perhaps not optimal for a very busy system. One potential solution is to allow the OCF to do dynamic load-balancing of sessions, replicating and tearing them down on additional cards based on their measured traffic, by maintaining session information internally. Asymmetric operations are easier to load balance, as they do not depend on the concept of the session. An additional benefit of implementing load-balancing in this way is that we can let the software driver handle small requests, reducing latency, and use the hardware producers for larger requests. One complication to this is that many cards (*e.g.*, Hifn) do not export internal state such as IVs or intermediate MAC results,

which makes such session sharing difficult.

Algorithm-chaining across cards. It is possible that an OCF consumer needs to chain together a number of cryptographic algorithms, but no hardware producer implements all these. Currently, this would cause the session to be established on the software pseudo-driver (which implements all algorithms). However, by maintaining session information inside the OCF, it is possible to create “virtual sessions” across multiple (hardware and software) producers. In this case, the OCF will issue multiple sequential requests to the various producers, invoking the consumer-specified callback routine at the end. We have a prototype of this, but we need to further evaluate the performance implications and trade-offs of doing multiple PCI transactions.

Asymmetric Multiprocessing (AMP) support. There is an increasing number of multi-processor systems. Most of these under-utilize the secondary processor, as many modern tasks are I/O-limited. Furthermore, it seems likely that the first version of SMP support for OpenBSD will be very coarse-grained: only one processor (and process) can be inside the kernel at a time. An alternative approach is to designate the secondary processor as a dedicated cryptographic accelerator that registers with the OCF as such. No special support by the OCF is necessary, and we are currently working toward an implementation of this.

OpenSSL support algorithm-chaining with OCF. As we mentioned in Section 4.2, TLS and SSH use the OCF at the granularity of the algorithm. That is, if both an encryption and a message authentication (MAC) algorithm have to be applied on an outgoing message, there will be two distinct calls to the OCF via `/dev/crypto`. (The same situation holds for incoming messages.) Since the OCF supports algorithm chaining, there is no reason why OpenSSL cannot take advantage of this to reduce the number of user/kernel crossings. This requires modification of the TLS implementation in OpenSSL and of OpenSSH, to support this algorithm chaining. While this is purely an implementation matter, the complexity of the OpenSSL code is a significant deterrent to progress in this direction.

Minimize number user/kernel crossings and data copying. In most practical uses of the OCF (especially in protocols like TLS or SSH), an application issues one or more crypto requests via `/dev/crypto`, followed by a `write()` or `send()` call to transmit the data. Similarly, a

read() or *recv()* call is followed by a number of requests to */dev/crypto*. This implies considerable data copying to and from the kernel, and potentially unnecessary context switching back and forth. An alternative approach is to “link” some crypto context to a socket or file descriptor (when doing application-level file encryption), such that data sent or received on that file descriptor are processed appropriately by the kernel: for example, a TLS implementation might construct a data record and simply *write()* it to the socket (one data copy and kernel crossing), only to have the kernel pass it to the OCF for processing before actually passing it on to TCP for transmission. This requires some discipline by the application, which must set the state on the socket and only *write()* appropriately-formatted record, as well as some support in the kernel to decode incoming TLS or SSH frames for processing by the OCF before passing them on to the application.

Another potential approach is to do “page sharing” of data buffers; when a request is given to */dev/crypto*, the kernel removes the page from the process’s address space and maps it in its own. When the request is done, the kernel re-maps the page back to the process’s address space, avoiding all data copying. This works well as long as */dev/crypto* remains a synchronous interface. If processes are allowed to have multiple pending requests, accesses to that page while it is being shared with the kernel must be caught and handled, similar to the way copy-on-write of memory pages is handled. An alternative is to block any process that tries to access such pinned-down pages until the crypto request is completed. Obviously, pages that are shared between processes can cause similar problems even in the current mode of operation. Operations that cross page boundaries also have to be dealt carefully.

7 Conclusions

We presented the OpenBSD Cryptographic Framework (OCF), a service virtualization layer implemented inside the kernel, that provides uniform access to cryptographic hardware accelerator cards by hiding card-specific details behind a carefully designed API. Other kernel subsystems and user-level processes can use the API with symmetric and asymmetric algorithms. The OCF offers several other features, such as load-balancing, session migration, and algorithm-chaining.

Our performance evaluation demonstrated the OCF’s ability to utilize available accelerators to within 95% of their peak performance. This validates our decision to design for ease of use by applications and seamless

support for new accelerators, over a device-specific approach which should be able to fully utilize that device’s capabilities. In addition, we demonstrated aggregate (across several concurrent applications) throughput for 3DES encryption in excess of 800 Mbps. Furthermore, use of hardware accelerators can remove contention for the CPU and thus improve overall system responsiveness and performance for unrelated tasks.

Our evaluation also allowed us to determine that the limiting factor for high-speed cryptography in modern systems is data copying and the PCI bus. Furthermore, small data-buffers should be processed in software, freeing hardware accelerators to handle larger requests that better amortize the system and PCI transaction costs. On the other hand, multi-threading results on increased utilization of the OCF, improving *aggregate* throughput. We made recommendations for future directions in architectural placement of cryptographic functionality, operating system provisions, and application design, and discussed several improvements and promising directions for future work.

Acknowledgements

Bob Beck and Markus Friedl helped with numerous OpenSSL integration issues we faced, since the “engine” code we required was unreleased. Bob also wrote the first working OpenSSL engine interfacing with */dev/crypto*. Markus helped with regression tests to ensure that */dev/crypto* operation was correct. Jonathan Smith and Sotiris Ioannidis provided valuable comments and insights. Sam Leffler adapted the OCF to the FreeBSD kernel. We would also like to thank Patrick McDaniel for providing high-quality shepherding of this paper.

We are grateful to Global Technologies Group, Inc. (GTGI) for providing us with two XL-Crypt (Hifn 7811) boards, one Hifn 6500 reference board, and one Hifn 7814 reference board. We are also grateful to Network Security Technologies, Inc. (NETSEC) for providing us with two Hifn 7751 boards, one Broadcom 5820 board, and two Broadcom 5805 boards. In addition, NETSEC funded part of the original development of the device-support software.

Part of this work was supported by DARPA and the Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-01-2-0537.

References

- [1] C. Adams. Independent Data Unit Protection Generic Security Service Application Program Interface (IDUP-GSS-API). RFC 2479, December 1998.
- [2] A. G. Broschius and J. M. Smith. Exploiting Parallelism in Hardware Implementation of the DES. In *Proceedings of the Crypto Conference*, pages 367–376, August 1991.
- [3] K. Claffy, G. Miller, and K. Thompson. The nature of the beast: Recent traffic measurements from an Internet backbone. In *Proceedings of the ISOC INET Conference*, July 1998.
- [4] C. Coarfa, P. Druschel, and D. Wallach. Performance Analysis of TLS Web Servers. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, February 2002.
- [5] T. de Raadt, N. Hallqvist, A. Grabowski, A. D. Keromytis, and N. Provos. Cryptography in OpenBSD: An Overview. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 93 – 101, June 1999.
- [6] D. Dean, T. Berson, M. Franklin, D. Smetters, and M. Spreitzer. Cryptology as a Network Service. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2001.
- [7] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson. Network subsystem design. *IEEE Network*, 7(4):8–17, July 1993.
- [8] D. C. Feldmeier and P. R. Karn. UNIX Password Security - Ten Years Later. In *Proceedings of the Crypto Conference*, pages 44–63, August 1990.
- [9] P. Gutmann. The Design of a Cryptographic Security Architecture. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.
- [10] P. Gutmann. An Open-source Cryptographic Coprocessor. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.
- [11] J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *Proceedings of the ACM SIGCOMM Conference*, pages 259–269, September 1993.
- [12] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.
- [13] A. D. Keromytis, J. Ioannidis, and J. M. Smith. Implementing IPsec. In *Proceedings of Global Internet (GlobeCom)*, pages 1948–1952, November 1997.
- [14] RSA Laboratories. *PKCS #11: Cryptographic Token Interface Standard, Version 2.01*, December 1997.
- [15] M. Lindemann and S. W. Smith. Improving DES Coprocessor Throughput for Short Operations. In *Proceedings of the 10th USENIX Security Symposium*, pages 67–81, August 2001.
- [16] J. Linn. Generic Security Service Application Programming Interface. RFC 2078, January 1997.
- [17] Microsoft Corporation. *Microsoft Cryptographic Application Programming Interface (CryptoAPI)*, second edition, December 1998.
- [18] S. Miltchev, S. Ioannidis, and A. D. Keromytis. A Study of the Relative Costs of Network Security Protocols. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 41–48, June 2002.
- [19] The Open Group. *Common Data Security Architecture (CDSA)*, second edition, May 1999.
- [20] C. Pu, H. Massalin, J. Ioannidis, and P. Metzger. The Synthesis System. *Computing Systems*, 1(1), 1988.
- [21] C. B. S. and J. M. Smith. Hardware/Software Organization of a High-Performance ATM Host Interface. *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)*, 11(2):240–253, February 1993.
- [22] J. M. Smith. Practical Problems with a Cryptographic Protection Scheme. In *Proceedings of the Crypto Conference*, pages 64–73, August 1990.
- [23] J. M. Smith and C. B. S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network*, 7(4):44–52, July 1993.
- [24] J. M. Smith, C. B. S. Traw, and D. J. Farber. Cryptographic Support for a Gigabit Network. In *Proceedings of INET*, pages 229–237, June 1992.
- [25] V. Smylov. Simple Cryptographic Program Interface (Crypto API). RFC 2628, June 1999.
- [26] Cross Organization CAPI Team. *Security Service API: Cryptographic API Recommendation, Updated and Abridged Edition*. National Security Agency, July 1997.

Appendix A: OCF Kernel API

- *int32_t crypto_get_driverid();*
int crypto_register();
int crypto_kregister();
int crypto_unregister();

Used by device drivers to register and unregister symmetric and asymmetric algorithm support with the OCF.

- *void crypto_done();*
void crypto_kdone();

Called by device drivers on completion of a request (symmetric and asymmetric, respectively).

- *int crypto_newsession();*

Called by consumers of cryptographic services (such as the IPsec stack) that wish to establish a new session with the framework. On success, the first argument will contain the Session Identifier (SID). The second argument contains all the necessary information for the driver to establish the session (keys, algorithms, offsets, *etc.* The third argument indicates whether only hardware acceleration is acceptable.

- *int crypto_freesession();*

Called to disestablish a previously-established session.

- *int crypto_dispatch();*

Called to process a request, encapsulated in its only argument. The various fields in that structure contain:

- The SID.
- The total length in bytes of the buffer to be processed,
- The total length of the result, which for symmetric crypto operations will be the same as the input length.
- The type of input buffer, as used in the kernel *malloc()* routine. This will be used if the framework needs to allocate a new buffer for the result (or for re-formatting the input).
- The routine that the OCF should invoke upon completion of the request, whether successful or not.
- The error type, if any errors were encountered. If the **EAGAIN** error code is returned, the SID has changed. The consumer should record the new SID and use it in all subsequent requests. In this case, the request may be re-submitted immediately. This mechanism is used by the framework to perform session migration (move a session from one driver to another, because of availability, performance, or other considerations).
- A bitmask of flags associated with this request. Currently, the only flag defined is **CRYPTO_F_IMBUF**, which indicates that the input buffer is an mbuf chain.
- The input and output buffers. The input buffer may be an mbuf chain or a contiguous buffer (as identified by the flags). The output buffer will be of the same type.
- A pointer to opaque data. This is passed through the crypto framework untouched and is intended for the invoking application's use.
- A linked list of operation descriptors, which indicate what operations should be applied, and in what sequence, to the input data. The descriptors indicate where each operation should start, the length of the data to be processed, where on the output buffer should the results be placed, the key material to be used, and various operation-specific flags (*e.g.*, what Initialization Vector to use for CBC-mode encryption).

- *int crypto_kdispatch();*

Similar to *crypto_dispatch()*, for public-key operations.

NCryptfs: A Secure and Convenient Cryptographic File System

Charles P. Wright, Michael C. Martino, and Erez Zadok
Stony Brook University

Abstract

Often, increased security comes at the expense of user convenience, performance, or compatibility with other systems. The right level of security depends on specific site and user needs, which must be carefully balanced. We have designed and built a new cryptographic file system called NCryptfs with the primary goal of allowing users to tailor the level of security vs. convenience to fit their needs. Some of the features NCryptfs supports include multiple concurrent ciphers and authentication methods, separate per-user name spaces, ad-hoc groups, challenge-response authentication, and transparent process suspension and resumption based on key validity. Our Linux prototype works as a stackable file system and can be used to secure any file system. Performance evaluation of NCryptfs shows a minimal user-visible overhead.

1 Introduction

Securing data is more important than ever. As the Internet has become more pervasive, security attacks have grown. Widely-available studies report millions of dollars of lost revenues due to security breaches [19]. Such concerns have prompted regulation efforts for the health-care (HIPAA [26]) and financial services (GLBA [16]) industries, as well as commitments from software vendors to provide better security facilities.

Yet, software to secure data files is not in wide use today. We believe one of the main reasons for this is that security software is not convenient to use: securing data files cannot be done easily and transparently. For security software to become universal, it has to balance several conflicting concerns: security, performance, and convenience. Whitten reported in 1999 that even experienced computer users could not use PGP 5.0 in less than 90 minutes and that one-quarter of the test subjects accidentally revealed the secret they were supposed to protect [28]. Work on security software in recent years has often focused on increasing the level of security and on performance, as reported in a recent comprehensive survey of storage systems security [21]; not much consideration has been given to user convenience [28]. It is not surprising that in recent years, prominent researchers such as Hennessey and Pike have advocated that the research community begin tackling difficult problems such as software usability [8, 18].

We have designed and developed an encryption file system (NCryptfs) whose primary goal is to ensure data confidentiality, while balancing security, performance and convenience. NCryptfs is a security wrapper that binds to a directory that stores ciphertext data. The ciphertext directory may be on any file system (e.g., EXT2 or NFS). Through NCryptfs, a cleartext view is presented via the standard UNIX file access API. We provide convenience by allowing administrators and users to customize the behavior of NCryptfs, while picking sensible defaults.

The threat models NCryptfs addresses include network sniffers, untrusted servers, and stolen machines. Normally, when exporting file systems over the network, cleartext data is sent over the network and the server must be trusted to keep the data confidential. When NCryptfs is deployed on the clients, only ciphertext file data is sent over the network, and the server does not have access to the cleartext data. Corporations and governments are storing more and more sensitive data on laptops, which are often stolen along with their valuable data. When NCryptfs is used, a stolen laptop will not reveal useful information to the thief. In both of these scenarios, NCryptfs attempts to restrict the information a compromised system reveals to an attacker to just the information that is actively being used.

NCryptfs is a successor to our much simpler encryption file system called Cryptfs [32]. Cryptfs was a proof-of-concept example of what useful features stackable file systems could offer. We used Cryptfs as a starting point and developed a comprehensive stackable cryptographic file system that offers new security and convenience features not available in Cryptfs. Both Cryptfs and NCryptfs were developed from our portable stackable file system toolkit called FiST [30, 33, 34]. Such stackable file systems can use any file system (e.g., EXT3, NFS, or CIFS) as the backing store for encrypted data. Some of the features that NCryptfs includes are:

- Support for multiple users, multiple keys, multiple ciphers, and multiple authentication methods (including challenge-response authentication between user processes and the kernel).
- Ad-hoc groups, allowing users to delegate “join” privileges to others, and for others to join or leave groups as needed.
- Per-process and per-session keys, with hooks for

processes to be informed of certain activities in NCryptfs (e.g., a request to re-authenticate).

- Key timeouts and revocation, which in addition to per-process keys allows us to suspend and resume processes based on key validity, as well as to add encryption transparently to unmodified programs that have already begun running.

In developing NCryptfs on Linux, we also noticed that a secure file system cannot be easily built as a standalone file system (stackable or native). The reason is that important file system information is accessed by other kernel components without consulting the file system. For example, we enhanced the Linux directory cache (dcache) and inode cache (icache) so that all accesses to cached (possibly cleartext) objects are validated first through NCryptfs. Additionally, we enhanced Linux's process management so that process destruction actions are coordinated with NCryptfs; the latter removes any related security info (e.g., keys and other objects) when a process or session terminates.

Many past secure file systems often made arbitrary decisions along a security-performance axes, with minimal consideration for user convenience [21]. NCryptfs was carefully designed so as to allow many levels of security and still offer ease-of-use and high performance; whenever possible, we allow administrators and users to select among several choices. Our performance benchmarks show NCryptfs's overhead to be just 5% for normal user activities.

The rest of this paper is organized as follows. Section 2 surveys background work. Section 3 describes the design of our system. We discuss interesting implementation aspects in Section 4. Section 5 presents an evaluation of our system. We conclude in Section 6.

2 Background

In this section we briefly describe other cryptographic file systems that provided the motivation for NCryptfs.

SFS SFS is an MSDOS device driver that encrypts an entire partition [6]. Once encrypted, the driver presents a decrypted view of the encrypted data. This provides the convenient abstraction of a file system, but relying on MSDOS is not secure because MSDOS provides none of the protection of a modern OS.

CFS CFS is a cryptographic file system that is implemented as a user-level NFS server [1]. It requires the user to create a directory on the local or remote file system to store encrypted data. The cipher and key are specified when the directory is first created. The CFS daemon is responsible for providing the owner access to the encrypted data via a special *attach* command. The daemon, after verifying the user ID and key, creates a directory in the mount point directory that acts as an un-

encrypted window to the user's encrypted data. Once attached, the user accesses the attached directory like any other directory. CFS is a carefully designed, portable file system with a wide choice of built-in ciphers. Its main problem, however, is performance. Because it runs in user mode, it must perform many context switches and data copies between kernel and user space.

TCFS TCFS is a cryptographic file system that is implemented as a modified kernel-mode NFS client. Since it is used in conjunction with an NFS server, TCFS works transparently with the remote file system, eliminating the need for specific attach and detach commands. To encrypt data, a user sets an encrypted attribute on directories and files within the NFS mount point [2]. TCFS integrates with the UNIX authentication system in lieu of requiring separate passphrases. It uses a database in */etc/tcfspwdb* to store encrypted user and group keys. Group access to encrypted resources is limited to a subset of the members of a given UNIX group, while allowing for a mechanism (called *threshold secret sharing*) for reconstructing a group key when a member of a group is no longer available.

TCFS has several weaknesses that make it less than ideal for deployment. First, the reliance on login passwords as user keys is not safe. Also, storing encryption keys on disk in a key database further reduces security. Finally, TCFS is available only on systems with Linux kernel 2.2.17 or earlier, limiting its availability.

BestCrypt BestCrypt is a commercially available loopback device driver supporting many ciphers [10]. Such a loopback device driver creates a raw block device with a single file, called a *container*, as the backing store. This device can then be formatted with any file system or used as swap space. Each container has a single cipher key. The administrator creates, formats, and mounts the container as if it were a regular block device. BestCrypt is ideal for single user environments but unsuitable for multiuser systems. In a single-user workstation, the user controls the details of creating and using a container. In a multi-user environment, however, the user must give the encryption key to a potentially untrustworthy administrator. Moreover, the ability to share containers among groups of users is limited, as BestCrypt gives different users equal rights to the same container. The Cryptographic Disk driver is similar to BestCrypt, and other loop-device encryption systems, but it uses a native disk or partition as the backing-store [4].

Cryptfs Cryptfs [32] is the stackable, cryptographic file system that serves as the basis for this work. It was never designed to be a secure file system, but rather a proof-of-concept application of FiST [34]. It supports only one cipher and implements a limited key management scheme.

EFS EFS is the Encryption File System found in Microsoft Windows, based on the NT kernel (Windows 2000 and XP) [14]. It is an extension to NTFS and utilizes Windows authentication methods as well as Windows ACLs [15,21]. EFS encrypts files using a long-term key. Encryption keys are stored on the disk in a *lockbox* that is encrypted using the user's login password. This means that when users change their password, the lockbox must be re-encrypted. If an administrator changes the user's password, then all encrypted files become unreadable.

StegFS StegFS is a file system that employs steganography as well as encryption [13]. If adversaries inspect the system, then they only know that there is some hidden data. They do not know the contents or extent of what is hidden. This is achieved via a modified EXT2 kernel driver that keeps a separate block-allocation table per *security level*. It is not possible to determine how many security levels exist without the key to each security level. Data is replicated randomly throughout the disk to avoid loss of data when the disk is mounted with an unmodified EXT2 driver and random blocks may be overwritten. Although StegFS achieves plausible deniability of data's existence, the performance degradation is a factor of 6–196, making it impractical for most applications.

3 Design

NCryptfs's primary design goals were to balance the often conflicting concerns of security, convenience, and performance [21]. Our two most important goals were security and convenience:

Security We ensure that the data stored using NCryptfs remains confidential, by using strong encryption to store data. We also modified the kernel to notify NCryptfs upon the death of a process and evict cleartext pages from the cache.

Convenience If a system is not convenient then users will not use it, or will circumvent its functionality [28]. The inconvenience of current cryptographic systems contributes to their lack of widespread adoption. NCryptfs makes encryption transparent to the application: any existing application can make use of strong cryptography with no modifications. We designed NCryptfs to be cipher agnostic, so it is not tied to any one cipher.

We also want the performance of our system to be as close as possible to a raw encryption operation as possible. Our final design goal was portability, which we achieve through the use of stackable file systems [33].

In Section 3.1 we describe the players in our system. In Section 3.2 we describe our key management. In Section 3.3 we describe the concept of an attachment, which

we use to provide much of the convenience associated with NCryptfs. In Section 3.4 we discuss ad-hoc groups. In Section 3.5 we discuss key revocation and timeouts. We describe system operation in Section 3.6.

3.1 Players

When designing any system, one of the most important questions is who are the players, or more simply who uses it. In NCryptfs we identified three groups of players: (1) the system administrator, (2) owners, and (3) readers and writers. We use the same taxonomy as Riedel, but add the system administrator [21].

System Administrator The system administrator originally mounts NCryptfs and must be able to enforce usage policies. The system administrator is trusted to properly install the NCryptfs kernel and user-space components. However, the system administrator is not trusted with encryption keys.

Owners The owner is the user who controls the encryption key for the data. The owner receives permissions (see Section 3.3) from the system administrator and may delegate them to other users.

Readers and Writers All other authorized users are either readers or writers. The only difference between an owner and a reader or writer is that the owner supplies the encryption key; all other readers and writers do not know the encryption key. An owner is implicitly a reader or writer, depending on the permissions that the system administrator delegates. For the system to be convenient, readers and writers must be able to use encryption transparently. Authorized readers and writers must also be able to delegate permissions received from other authorized readers and writers.

Any user who attempts to exceed their delegated permissions is considered an adversary.

3.2 Key Management

The security of encrypted data is only as strong as the policy that is put in place to protect the keys. NCryptfs makes the assumption that the underlying storage media can be read and tampered with, so to ensure data confidentiality, it must be encrypted. Before the encrypted data is used, the owner must provide the key to NCryptfs (presently by entering a passphrase). Once the key is sent to kernel space, NCryptfs stores it in core memory. NCryptfs will use the encryption key on behalf of readers and writers, without revealing it to them. When cryptographic algorithms are used for authentication, authentication information is distinct from the encryption key. After the initial authentication takes place, the result is bound to a specified user, group, session, or process.

NCryptfs uses a long-lived key to encrypt all data and file names written to disk. If we used a short-lived key, then whenever the key changed, all data would have to

be re-encrypted. To avoid this performance penalty, we use long-lived keys. NCryptfs uses the underlying file system to store ciphertext data, but all other data related to the encryption key is stored in pinned core memory that can not be swapped to disk.

NCryptfs is cipher agnostic. It uses cipher modules that are treated as simple data transformations. The only requirement that NCryptfs makes of the cipher is that it must be able to encrypt an arbitrary length buffer into a buffer of the same size. Most widely-used ciphers are able to do this in *Cipher Feedback Mode* (CFB) [25]. CFB mode allows us to keep the size of encrypted files the same. Changing the size of files complicates stackable file systems and decreases performance [31]. Selecting an appropriate cipher allows the user to select where they want to lie on the security-performance-convenience continuum. If the user is more concerned about performance, then a faster but less secure cipher may be chosen (e.g., one with a shorter key length). This also affects convenience: if the cipher is too slow then the user may not use encryption at all.

3.3 Attachments

We associate each encryption key with an *attach*. Attachments, inspired by CFS, allow owners to have personal encrypted directories [1]. An attach is much like an entirely separate instance of a stackable file system. Each attach has a corresponding directory entry within the NCryptfs mount point and stacks on a different lower-level directory. This relationship can be seen in Figure 1. There are three attachments: each attach is a directory entry within `/mnt/ncryptfs` and stacks on a separate lower directory. In this figure the three attaches are `proj`, `mcm`, and `cpw`—which stack on `/proj/src`, `/home/mcm/enc`, and `/home/cwright/mail`, respectively. Encrypted files are stored within the directories `/proj/src`, `/home/mcm/enc`, and `/home/cwright/mail`. The plaintext view of these files is available through `/mnt/ncryptfs/proj`, `/mnt/ncryptfs/mcm` and `/mnt/ncryptfs/cpw`, respectively.

An attach can be thought of as a lightweight user-mode mount. Unlike a regular mount, an NCryptfs attach is not a dangerous operation that only superusers can perform safely. A mount may hide data by mounting on top of a non-empty directory, but an attach can not hide any data because NCryptfs does not allow any files or directories to be created in the root of the NCryptfs file system. A mount may introduce new, possibly dangerous data, such as devices or `setuid` programs, but NCryptfs only presents an unencrypted view of the existing data in the system, without modifying metadata. Since an attachment does not hide data or introduce new

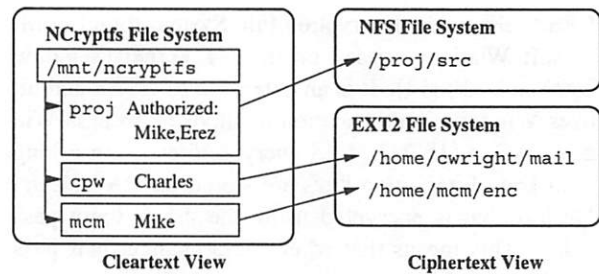


Figure 1: Attach Mode Mounts. This example shows three attachments. Each attach is a directory entry within `/mnt/ncryptfs`.

data, unlike a mount, it is a safe operation.

There are compelling reasons to use an attach to achieve this behavior, rather than simply mounting NCryptfs multiple times. In general, only the superuser may mount a new file system. It is possible to allow a user to mount a specific file system with specific parameters, but if we want to allow users to encrypt arbitrary data then this does not suffice, because `/etc/fstab` entries need to be created for each encrypted directory. Finally, most UNIX operating systems have a hard limit on the number of mounts that are allowed; or the OS uses per-mount data structures that do not scale well (e.g., linked lists). Using an attach permits the use of a specific type of stackable enhancement for many lower directories without running into hard limits or degrading system performance for other operations. NCryptfs uses the directory cache (dcache) to store attaches, because in Linux the dcache organizes many entries efficiently.

Attaches were originally designed for convenience. However, separating the name space for each encryption key also provides several other benefits in the context of stackable file systems. Foremost among these benefits is that the dcache can not handle two different views of an encrypted file system. For example, the situation where `cpw` has `/mnt/ncryptfs/foo` encrypted with one key, and user `ezk` has `/mnt/ncryptfs/foo` encrypted with another key is possible without attaches—since `foo` may encrypt to different ciphertexts with different keys. Having two files with the same name causes their data to get intermixed within the dcache and page cache. Even if the data can not be read, knowing the existence of a given file may provide valuable information to an attacker. The better way to allow multiple users to concurrently use a single cryptographic file system is to separate the name space. The only name space mixing using attach mode is the name of the attach (which must be unique). This means that NCryptfs has a completely separate name space for each set of encrypted files.

Each attach has private data that is relevant only to that specific attachment. The per-attach data is made up of an encryption key, authorizations (access control en-

tries), and active sessions. These three data structures separate encryption, authorization, and active sessions. These three data structures model flexible and diverse policies including ad-hoc groups:

Encryption Key This information is specific to the cipher for this attach. This data includes the encryption key and any information (such as initialization vectors) required to perform encryption. NCryptfs passes this data to each encryption or decryption operation, but has no knowledge about the contents of this data. The cipher is wholly responsible for its maintenance and interpretation. This data is opaque to NCryptfs so that a multitude of ciphers can be used without any modifications to NCryptfs.

Authorizations Each attach has one or more *authorizations*. An authorization gives an entity access to NCryptfs after the entity meets a certain authentication criteria. An entity may be a process, session, user, or group. The authentication criteria consists of a method (e.g., password) and data that is specific to this method (e.g., a salted hash of the password).

Active Sessions Each attach also has one or more *active sessions*. An active session contains the description of an entity and the permissions granted to that entity. Note that NCryptfs active sessions are not necessarily the same as UNIX sessions (e.g., an active session can be bound to a user or process). Once an entity has authenticated according to the rules in an authorization, an active session is created. One authorization can map to multiple active sessions (e.g., a user authenticates in two sessions using a single authorization entry). Each active session corresponds to an authorization that exists or existed in the past. If an authorization is removed, the active sessions are allowed to remain (revocation is discussed in Section 3.5).

To ensure maximum flexibility, NCryptfs uses fine-grained permissions. Each authorization and active session contains a bitmask of permissions. NCryptfs permissions are the standard read, write, and execute bits that UNIX already defines plus an additional seven operations:

- **Detach** allows removal of the attachment from NCryptfs. When users have completed their work, the detach operation ensures that all resources (including keys) are freed.
- **Add an Authorization** allows users to delegate a subset of their permissions to new authorizations. By default, only the session that created the attach is authorized. This allows an owner to work with multiple sessions (e.g., using two different xterms) or to give other users permission to use the attach. Using a UNIX session identifier makes it more dif-

ficult to hijack an authentication, whereas a UID can be easily changed using `/bin/su`.

- **List Authorizations** allows users to verify and examine which entities (users, sessions, processes, and groups) are authorized to use this attach. Sensitive information (such as the authentication criteria) is not returned.
- **Delete an Authorization** allows users to remove an authorization from an attach.
- **Revoke an Active Session** allows users to prevent a currently-authenticated user from accessing NCryptfs. This can be combined with the authorization-deletion operation to prevent any future use of an attach.
- **List Active Sessions** allows users to verify and examine which users have authenticated to an attach.
- **Bypass VFS Permissions** allows users to take on the identity of the file's owner for files within the attach. This permission is required to implement ad-hoc groups, which allow the convenient sharing of encrypted data (see Section 3.4).

3.3.1 Attach Access Control

By default, any user is allowed to create an attachment with full permissions (except bypass VFS permissions). The system administrator can change this default policy by adding authorizations to the NCryptfs mount point. Each authorization allows a single entity to attach or authenticate to an attach. The main NCryptfs mount point has no active sessions, only authorizations. The mount point can not require authentication, because authentication takes place through an `ioctl`. If the user has not already been granted permission, then the `ioctl` will not be permitted. Once the attach or authentication takes place, the entity receives a subset of permissions in the authorization. Authorizations for the NCryptfs mount point require two additional permissions:

- **Attach** allows a user to create an attach.
- **Authentication** allows a user to authenticate to an attach.

The system administrator can also limit the maximum numbers of attaches and the maximum and minimum key timeouts both on a global and on a per user basis.

3.3.2 Attach Names

There are three methods to generate attach names. First, when a user attaches, by default the user is allowed to choose the name for their attach. This allows a convenient name that the user can easily remember and type, but this name may reveal information about the contents and multiple users may want to use the same name (e.g., `mail` or `src`).

In the second method, NCryptfs generates a name

based on the entity doing the attaching to prevent name space collisions between users. We do this by appending a one-letter prefix based on the entity type to the entity's numeric identifier. For example, u500 represents UID 500, and u500s200 represents UID 500 with session ID 200. These names are easy to remember, but reveal who is doing the attach. This method is less convenient than allowing users to choose their own names, and may not be unique since each entity can have multiple attaches (e.g., UID 500 may have several encrypted directories, all used in session ID 200).

The third method is to randomly generate unique attach names. This allows a specific user to have multiple generated attaches. This method has the added benefit that it enforces using names for attaches that do not reveal information about the attachments' contents. NCryptfs guarantees randomly generated names that have no name space collisions. These names reveal no information about the contents, but are harder to remember and type. This method is even less convenient to users, but guarantees unique names.

3.4 Groups

NCryptfs supports native UNIX groups like any other entity. A UNIX group has some disadvantages, primarily that a group needs to be setup by the system administrator ahead of time. This means that users must contact the system administrator, and then wait for action to be taken.

NCryptfs supports ad-hoc groups by simply adding authorizations for several individual users (or other entities). The problem with this approach is that each additional user must have permissions to modify the lower level objects, since NCryptfs by default respects the standard lower-level file system checks. If the permissions on the lower level objects are relaxed, then new users can modify the files. However, without a corresponding UNIX group, it is difficult to give permissions to precisely the subset of users that must have them. If the permissions are too relaxed, then rogue users can trivially destroy the data and cryptanalysis become easier—even without the system being compromised.

NCryptfs's `bypass-VFS-permissions` option solves this problem. The owner of the attach can delegate this permission (assuming root has given it to the owner). When this is enabled, NCryptfs performs all permission checks independently of the lower-level file system. This allows NCryptfs to be used for a wider range of applications. This feature is described in depth in Section 4. When ACLs become common, they can be used in place of this mechanism [5, 9].

3.5 Timeouts and Revocation

Keys, authorizations, and active sessions all can have a timeout associated with them. When an object times out, NCryptfs executes a user-space program optionally specified at attach time. For example, the user may specify an application that ties into a graphical desktop environment to prompt for the user's passphrase.

NCryptfs takes one of four actions when a timed-out object is referenced:

- All further file system operations fail with "permission denied." This policy is strict and secure, but it is inconvenient.
- Opening a file fails, but already open files continue to function. This is useful because it allows existing work to complete.
- Files that are already open continue to function, but when a user attempts to open a new file, the process is put to sleep until the operation can succeed (e.g., the user re-authenticates). This also allows old work to complete, but no operations will immediately fail, so users do not need to sort out as many partial completions and errors.
- All operations cause the process to be put to sleep until the operation can succeed: open, read, write, etc. block until re-authentication. This prevents even open files from being accessed until the user re-authenticates, and is convenient because no operation will fail until the user has had a chance to re-enter the passphrase. This is similar to the authentication timeout employed in Zero-Interaction Authentication [3].

After the user re-authenticates, blocked processes wake up (or successfully complete operations). If the system is configured to cause all operations to fail or cause all processes to go to sleep, the key timeout deletes the key from memory. If existing files are permitted to continue functioning, then the key must remain in memory, but NCryptfs prevents new files from accessing it.

An authorization timeout prevents new users from authenticating with that authorization, but active sessions may continue to use the attach. This can be used to create login windows, such that all logins must take place between 9:00AM and 10:30AM.

An NCryptfs kernel thread wakes up sleeping processes after a user-specified duration. The function that caused NCryptfs to put the process to sleep returns an error. This prevents processes from waiting indefinitely for an event that may never occur (e.g., a re-authentication).

Active sessions can be revoked. A timeout is a special case of a revocation because it is a scheduled revocation, so an active session revocation has the same behavior as an active session timeout with one key difference. If

an active session times out, then it may be indefinitely extended by re-authenticating even if the corresponding authorization was removed. When an active session is revoked, it may not be re-enabled. If manual intervention was taken by the owner to prevent a user from accessing NCryptfs, then we can not safely allow the user to undo that operation.

3.6 System Operation

In this section we describe a typical scenario for NCryptfs usage. The end product of the following example is the structure seen in Figure 1. We show the steps that lead up to the attach structure that is in place.

First, the system administrator mounts NCryptfs on `/mnt/ncryptfs`, then adds authorizations for Mike, Erez, and Charles. User Erez has full permissions including bypass VFS permissions, and other users have full permissions except bypass-VFS-permissions. No data is encrypted just yet. The mount point only contains the `."` and `."` directory entries.

Next, Charles, Mike, and Erez create attaches. For example, Charles runs the command `"nc_attach -c blowfish /mnt/ncryptfs cpw /home/cwright/mail"`. Then, `nc_attach` prompts for a key. The key may be entered as a hexadecimal or ASCII string. An encryption key is then derived from this string using PKCS#5 PBKDF2 [23]. The key is passed to the Blowfish cipher module [25]. After this command is completed, `/mnt/ncryptfs/cpw` presents a decrypted view of the files in `/home/cwright/mail`. In this situation, Charles may access the encrypted files and he has full permissions. The mapping between `mcm` and `/home/mcm/enc` is created in the same manner.

Erez performs the same operation to map `/mnt/ncryptfs/proj` to `/proj/src`. Erez wants to give Mike the ability to read files in this attach as well. Mike and Erez share no UNIX group among themselves, but Erez has the ability to bypass VFS permissions. Erez adds an authorization for Mike with the `bypass-VFS-permissions` option enabled. To create this authorization, Erez specifies several options aside from the permissions. The first, required setting, is the authorization criteria. In this example password authentication was chosen. A salted MD5 hash of the passphrase is passed to the kernel when creating a password authorization [22]. Using a salted MD5 hash allows Mike to choose his passphrase without revealing it to Erez, and Erez can store it in a configuration file without the original passphrase being revealed. Additionally, Erez may authorize any session that Mike opens, but to use the attach from a session, Mike needs to be authenticated in that particular session. This ties an active session to a specific virtual terminal, to make hijacking the active session more difficult.

Finally, Erez can specify timeouts for the authorization that he is creating. In this example, we chose an authorization timeout of six hours, an active session timeout of one hour, and an inactivity timeout of fifteen minutes. This means that Mike can authenticate to the attach within the next six hours; once authenticated, he can use the attach for one hour without re-authenticating; and if he does not use the encrypted files for more than fifteen minutes, he must re-authenticate. After a timeout, all of Mike's processes go to sleep until Mike re-authenticates. This shrinks the window in which the encryption key may be used.

To use the attach that Erez has created, Mike runs `nc_auth /mnt/ncryptfs proj`, and `nc_auth` prompts him for a passphrase. If Mike is successful, then he may use the files in `/mnt/ncryptfs/proj`. Mike starts a large compile, which lasts for more than an hour. After an hour, the active session timeout is triggered. Mike has configured a user-space hook for his timeout; this hook loads a small X11 program which prompts him for his passphrase, and re-authenticates to NCryptfs. After Mike successfully enters his passphrase, his compile resumes. If Mike is unable to enter the passphrase, then after a configurable amount of time the compile fails with a "permission denied" message.

4 Implementation

We implemented a prototype of NCryptfs on Linux 2.4.18 using FiST, a language for stackable file systems, as a starting point [33]. Although much of NCryptfs was implemented as a standalone stackable file system, we needed to modify some parts of the kernel to increase the security of NCryptfs. Using stackable mechanisms allows us to create new file systems without changes to the rest of the OS. However, Linux is not sufficiently flexible to allow the creation of completely independent secure file systems. Although this reduces portability as compared to a standalone stackable file system, the increased security is worth this trade off. We have designed these features in a way that NCryptfs can be used without them (at the cost of reduced security). In this section, we discuss four interesting aspects of our implementation: on-exit callbacks for tasks, cache cleaning, the imperfect stacking characteristics of permission handling in Linux, and our use of cryptography.

On-exit callbacks Linux does not provide a way for kernel components to register interest in the death of a process. We extended the Linux kernel to allow an arbitrary number of private data fields and corresponding callback functions to be added to the per-process data structure, `struct task`. When a process exits, the kernel first executes any callback functions before destroying the task object. NCryptfs uses such callbacks

in two ways. First, when processes die, we immediately remove any active sessions that are no longer valid (e.g., if the last process in a session dies, we remove the NCryptfs active session). The alternative to on-exit callbacks would have been to use a separate kernel thread to perform periodic garbage collection, but that leaves security data vulnerable for an unacceptably long window. If an authenticated process terminates and the active session remains valid, then an attacker can quickly create many processes to hijack the active session entry. Using the on-exit call back, the information is invalidated before the process ID can be reused. Also with on-exit callbacks we release memory resources as soon as they are no longer needed.

The second use for private per-process data is in NCryptfs's challenge-response authentication ioctl, which proceeds as follows. First, the user process runs an ioctl to request a challenge. We generate a random challenge, store it as task-private data, and then send the challenge's size back to the user. Second, the user allocates a buffer and calls the ioctl again. This time the kernel actually returns the challenge data. The user performs some function on the data (e.g., HMAC-MD5 [12]), that requires some piece of secret knowledge to transform the data. Finally, the user program calls the ioctl a third time with the response. If the response matches what the kernel expects, then the user is authenticated. Using the task private data sets up a transaction between the kernel and a task. Even though there are several ioctls in this authentication sequence, it is no different than if a process had authenticated itself using a single ioctl. The challenge and its size are not useful to an attacker, since the challenge can only be used for a single authentication attempt. Only the last ioctl call modifies the state of the task. Finally, if an authentication is aborted, then the challenge is discarded on process termination.

Cache Cleaning Cleartext pages normally exist in the page cache. Unused file data and metadata may remain in the dcache and icache, respectively. If a system is compromised, then this data is vulnerable to attack. For example, an attacker can examine memory (through `/dev/kmem` or by loading a module). To limit this exposure, NCryptfs evicts cleartext pages from the page cache, periodically and on detach. Unused dentries and inodes are also evicted from the dcache and icache, respectively. For added security at the expense of performance, NCryptfs can purge cleartext data from caches more often. In this situation, the decryption expense is incurred for each file system operation, but I/O time is not increased if the ciphertext pages (e.g., EXT2 pages) are already in the cache. Zero-Interaction Authentication (ZIA), another encryption file system based

on Cryptfs, takes the approach of encrypting all pages when an authentication expires [3]. This is less efficient than the NCryptfs method, because ZIA will also maintain copies of pages in the lower-level file system. ZIA requires the initial encryption of pages, and will use memory for these encrypted pages.

Bypassing VFS Permissions When intercepting permissions checks, it is trivial to implement a policy that is more restrictive than the underlying file system's normal UNIX mode-bit checks. To support ad-hoc groups without changing lower-level file systems, however, NCryptfs needed to completely ignore the lower-level file system's mode bits so that NCryptfs could implement its own authentication checks and yet appear to access the lower-level files as their owner.

The flow of an NCryptfs operation that must bypass VFS permissions (e.g., `unlink`) is as follows:

```
sys_unlink {                               /* system call service routine */
    vfs_unlink {                             /* VFS method */
        call nc_permission()
        if not permitted: return error
        nc_unlink {                         /* NCryptfs method */
            call nc_perm_preop()            /* code we added */
            vfs_unlink {                   /* VFS method */
                call ext2_permission()
                if not permitted: return error
                call ext2_unlink()          /* EXT2 method */
            }                               /* end of inner vfs_unlink */
            call nc_perm_fixup()            /* code we added */
        }                                  /* end of nc_unlink */
    }                                       /* end of outer vfs_unlink */
}
```

The VFS operation (e.g., `vfs.unlink`) checks the permission using the `nc_permission` function. If the permission check succeeds, the corresponding NCryptfs-specific operation is called (e.g., `nc_unlink`). NCryptfs locates the lower-level object and again calls the VFS operation. The VFS operation checks permissions for the lower-level inode before calling the lower-level operation. This control flow means that we can not actually intercept the lower-level permission call. Instead, we change `current->fsuid` to the owner of the lower-level object before the operation is performed and restore it afterward, which is done in `nc_perm_preop` and `nc_perm_fixup`, respectively. We change only the permissions of the current task, and the process can not perform any additional operations until we restore the `current->fsuid` and return from the NCryptfs function. This ensures that only one lower file system operation can be performed between `nc_perm_preop` and `nc_perm_fixup`.

Linux 2.6 will have Linux Security Modules (LSMs) that allow the interception of many security operations [29]. Unfortunately, the LSM framework is not sufficient to bypass lower-level permissions either. Their VFS calls the file-system-specific permission operation

first. The LSM permissions operation is called only if the file-system-specific operation succeeds. The LSM operation can allow or deny access only to objects that the file system has already permitted. A better solution is to consult the file-system-specific permission operation. This result should be passed to the LSM module which can make the final decision, possibly based on the file-system-specific result.

Cryptography To ensure data confidentiality, NCryptfs uses strong cryptography algorithms (e.g., Blowfish or AES in CFB mode). File data and file names are handled in two different ways.

Data is encrypted one page at a time, using an initialization vector (IV) specified along with the encryption key XORed with the inode number and page number. For security, ideally the entire file would be encrypted at once, but then random access would be prohibitively expensive; to access the n th byte of data, n bytes would need to be decrypted, and any write would require re-encryption of the entire file. For optimal performance, each byte would be encrypted individually, but without data interdependence, encryption becomes significantly less secure [24].

File names are encrypted with the IV XORed with the inode number of the directory, but the output may contain characters that are not valid UNIX pathnames (i.e., / and NULL). To rectify this problem, the result is base-64 encoded before being passed to the lower-level file system. This reduces the maximum path length by 25%. A checksum is stored at the beginning of the encrypted file name for two reasons. First, if a file name is not encrypted with the correct key, then this checksum will prevent it from appearing in NCryptfs. Second, since CFB mode is used, if two files have a common prefix, then they will have a common encrypted prefix. Since it is unlikely that these two files will have the same checksum, prefixing their names with the checksum will prevent them from having the same prefix in the ciphertext. Finally, the directory entries “.” and “..” are not encrypted to preserve the directory structure on the lower-level file system.

5 Evaluation

We developed a prototype of NCryptfs in Linux 2.4.18. We compare it with CFS, TCFS, and BestCrypt. We chose these three different systems because they represent a cross section of techniques:

- **CFS** is a localhost NFS server, and among the first widely-used cryptographic file systems.
- **TCFS** is an NFS client that implements cryptographic functionality including data integrity assurance.

- **BestCrypt** is an encrypted loopback device driver. BestCrypt is a commercial product.
- **NCryptfs** is our stackable file system.

We begin by describing the various features, security, and convenience aspects of each system in Section 5.1. We compare their performance in Section 5.2.

5.1 Feature Comparison

In this section we present a comparison of the different functionality implemented by CFS, TCFS, BestCrypt, Cryptfs (the predecessor to NCryptfs), and NCryptfs. We identified the following metrics, which we summarized in Table 1:

1. **No keys stored on disk:** If keys are stored persistently, then encryption adds little security. All of the systems we compared, except TCFS, do not store keys on disk. TCFS stores each user's key in a database encrypted using the user's login password. Login passwords are restricted to eight characters, must contain only printable characters, and are often used and thus may be inadvertently exposed in cleartext. If separate passwords were used, then this would not be a weakness in TCFS.
2. **Keys protected from swap devices:** If memory becomes scarce, memory that could contain keys may be swapped. NCryptfs prevents this by pinning keys in physical memory, but cleartext process data can still be written to swap. An encrypted swap partition can prevent all sensitive data (and non-sensitive data) from being written to persistent media in the clear [20]. BestCrypt can encrypt an entire swap device to prevent data from leaking.
3. **Reveals no directory structure information:** CFS, TCFS, and NCryptfs reveal the number of files and their structure as well as inode meta-data information. BestCrypt uses a single file for an entire encrypted file system so it does not reveal this information.
4. **Multiple Concurrent Users:** CFS and NCryptfs allow users to create their own personal attachments. NCryptfs additionally allows multiple users to use a single attachment, with distinct permissions. TCFS allows multiple users to use a common name space with different keys. BestCrypt allows different users to use the same container with different passwords, but with the same permissions.
5. **Users do not need root intervention:** After an initial setup, users do not need root intervention to create new sets of encrypted data in CFS or NCryptfs. In TCFS, the system administrator must run `tcfsadduser` for each user who needs TCFS access. In BestCrypt and Cryptfs, root must allow each encrypted directory to be mounted.

	Feature	CFS	TCFS	BestCrypt	Cryptfs	NCryptfs
1	No keys stored on disk	✓	^a	✓	✓	✓
2	Keys protected from swap devices			✓ ^b		✓
3	Reveals no directory structure			✓		
4	Multiple concurrent users	✓ ^c	✓	✓		✓
5	Users do not need root intervention	✓				✓
6	Multiple ciphers	✓	✓	✓		✓
7	Automatic cipher loading		✓			✓
8	Separate permissions per user					✓
9	Group support – UNIX GID		✓			✓
10	Group support – ad-hoc					✓
11	Challenge-response authentication					✓
12	Data integrity assurance		✓			
13	Per-file encryption flag		✓			
14	Threshold secret sharing		✓			
15	Key timeouts	✓				✓
16	User-space timeout callback					✓
17	Process sleep/wakeup on key timeout					✓
18	Implementation technique	NFS server	NFS client	loop device	stackable	stackable
19	No. of systems available	any UNIX	3 ^d	2	3	1 ^e
20	Additional Blowfish LOC (Lines Of Code, excludes cipher implementation)	33	109	99	0	76
21	Total core LOC	5258	14731	3526	4943	6537

Table 1: Feature comparison. A check mark indicates that the feature is supported, otherwise it is not.

^aTCFS stores on disk keys encrypted with login passwords.

^bBestCrypt can encrypt the entire swap device.

^cCFS supports multiple users, but is single threaded.

^dTCFS provides cipher modules and data integrity assurance only on Linux.

^eNCryptfs is based on the FiST templates, which are available on three systems.

6. **Multiple Ciphers:** CFS, TCFS, BestCrypt, and NCryptfs support this.
7. **Automatic cipher loading:** TCFS and NCryptfs support this feature. In CFS, all ciphers are statically compiled into `cfsd`. BestCrypt loads all available ciphers, whether they are used or not.
8. **Separate permissions per user:** When using groups, NCryptfs allows each member to have individually-defined permissions (e.g., read, write, or detach). Other systems treat all users the same as each other.
9. **Group support – UNIX GID:** TCFS and NCryptfs support UNIX groups.
10. **Group support – ad-hoc:** Only supported in NCryptfs.
11. **Challenge-response authentication:** Only supported in NCryptfs.
12. **Data integrity assurance:** TCFS detects modifications to the ciphertext. If data is modified on the underlying file system, then CFS, NCryptfs, and BestCrypt do not detect this.
13. **Per-file encryption flag:** TCFS allows users to specify whether data is encrypted on a per-file basis. This may confuse users, since not all files within TCFS are be encrypted.
14. **Threshold secret sharing:** TCFS allows a group key to be split into n pieces. If m of these n members of the group insert their key into TCFS, then the full key can be reconstructed.
15. **Key timeouts:** CFS can automatically detach an attach after a certain period of time. NCryptfs can time out keys, active sessions, and authorizations.
16. **User-space timeout callback:** NCryptfs can optionally execute a user-space program on timeouts.
17. **Process sleep/wakeup on key timeout:** NCryptfs has four types of possible behavior on timeouts: all operations fail, new files operations fail, all operations put the calling process to sleep, or operations on new files put the calling process to sleep.
18. **Implementation technique:** CFS is a user-space localhost NFS server that works with standard NFS clients. Running in user-space decreases performance, but increases portability. TCFS is a kernel-space NFS client that works with any NFS server. BestCrypt is a kernel loopback device driver. This means that it has lower overhead than other sys-

tems. Cryptfs and NCryptfs are stackable file systems that run in kernel space. Since stackable file systems run in kernel space, they have better performance than user-space file systems, and are easier to develop than disk or network-based file systems.

19. **Number of systems available:** CFS can run on any UNIX system. TCFS runs on Linux, OpenBSD, and NetBSD, but is only feature complete on Linux. BestCrypt runs on Linux and Windows. Cryptfs runs on Linux, FreeBSD, and Solaris. The NCryptfs prototype runs on Linux, but is based on the FiST templates, which run on Linux, FreeBSD, and Solaris.
20. **Additional Blowfish Lines Of Code (LOC):** The total number of lines of code needed to interface with an existing cipher is a good metric for how difficult it is to add additional ciphers. To interface with Blowfish, CFS, TCFS, BestCrypt, and NCryptfs use small wrappers. Cryptfs hard-codes the calls to Blowfish.
21. **Total core LOC:** The number of LOC in the file system is a good measure of maintainability, complexity, and the amount of initial effort to write the system. CFS, Cryptfs, and NCryptfs have roughly the same number of LOC. TCFS re-implements an NFS client and is more than twice the size of any other system. BestCrypt has the smallest implementation, which is to be expected because it is a loopback device driver, not a file system.

NCryptfs supports a rich feature set that allows system administrators and users to tailor it to their site-specific security, performance, and convenience needs.

5.2 Performance Comparison

We compared the performance of CFS, TCFS, BestCrypt, and NCryptfs. We ran all benchmarks on a 1.7GHz Pentium 4 machine with 128MB of RAM. All experiments were located on a 30GB 7200 RPM Western Digital Caviar IDE disk formatted with EXT2. The machine was installed with Red Hat Linux 7.3. For CFS, BestCrypt, and NCryptfs, we ran a vanilla 2.4.18 kernel. For TCFS, we used the most recent supported kernel, 2.2.17, with the TCFS patch applied. To ensure cold cache, we unmounted the file systems where the experiments took place between each test. All other executables and libraries (e.g., compilers) were located on the root file system. We ran all tests several times, and our computed standard deviations were less than 5%. We chose Blowfish with a 128 bit key for our cipher, since it is widely available and performs well in software. We recorded elapsed, system, and user times for all tests.

5.2.1 Configurations

We used the following ten configurations:

- **EXT2-24** A vanilla EXT2 running on the 2.4.18 kernel. It serves as a baseline for performance of other configurations.
- **EXT2-22** A vanilla EXT2 running on the 2.2.17-tcfs kernel. It serves as a baseline for TCFS performance.
- **CFS-NULL** CFS using the identity cipher (this copies data with no modification). This demonstrates the overhead of the file system without cryptographic operations.
- **TCFS-NULL** TCFS using the identity cipher.
- **BC-NULL** BestCrypt using the identity cipher.
- **NC-NULL** NCryptfs using the identity cipher
- **CFS-BF** CFS using the Blowfish cipher. This demonstrates the overhead of CFS including cryptography.
- **TCFS-BF** TCFS using the Blowfish cipher. TCFS is designed to generate several keys per file. However, due to a memory leak we discovered in TCFS, we were forced to use a single key for all encryption operations. This means that the overhead introduced by TCFS will be underrepresented because initializing a Blowfish key is an expensive operation, which uses 4168 bytes of memory and requires 521 iterations of Blowfish encryption [25].
- **BC-BF** BestCrypt using the Blowfish cipher.
- **NC-BF** NCryptfs using the Blowfish cipher.

5.2.2 Workloads

We tested our configurations using two workloads: one CPU-intensive and another that is I/O intensive.

The first workload was a build of Am-utils [17]. We used Am-utils 6.0.7: it contains over 50,000 lines of C code in 425 files. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 265 additional files. The Am-utils compile contains a fair mix of file system operations; it is CPU intensive and performs many meta-data operations during the `configure` process. This workload demonstrates what type of performance impact a user may see while using NCryptfs.

The second workload we chose was Postmark [11]. We configured Postmark to create 20,000 files and perform 100,000 transactions in ten directories. This benchmark uses little CPU, but is I/O intensive. Postmark focuses on stressing the file system by performing a series of file system operations such as directory lookups, creations, and deletions on small files. A large number of small files is common in electronic mail and news servers where multiple users are randomly modifying small files. We chose the above parameters for the number of files and transactions as they are typically used and recommended for file system benchmarks [11, 27].

5.2.3 Am-utils Results

The elapsed time overhead for Am-utils is shown in Figure 2 and the first two rows of Table 2. This shows that CFS and TCFS performance suffer from using the network stack. TCFS performance additionally suffers from data integrity checks. BestCrypt uses a kernel thread to perform all encryptions, so cryptographic operations may continue after the termination of the instrumented process. This helps to improve perceived performance, which we further explore later in Section 5.2.4. NCryptfs only minimally impacts performance.

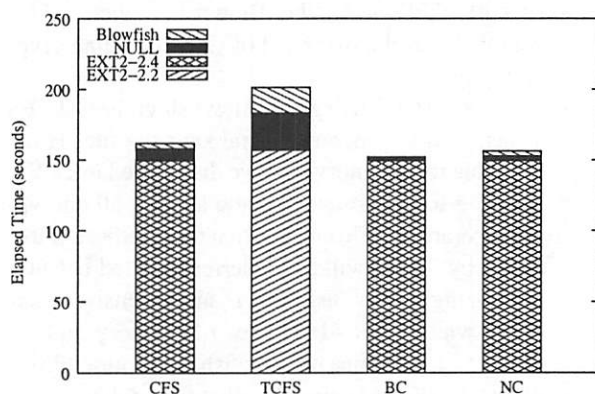


Figure 2: Am-utils build elapsed time. Each bar represents an EXT2 configuration, a NULL configuration, and an encrypting configuration.

Configuration	CFS	TCFS	BC	NC
Elapsed Time – NULL	5.7	16.9	1.5	2.2
Elapsed Time – BF	8.4	28.4	1.7	4.5
System Time – NULL	25.5	50.3	0.7	4.6
System Time – BF	39.5	93.7	1.8	17.0

Table 2: Am-utils percentage overheads over EXT2

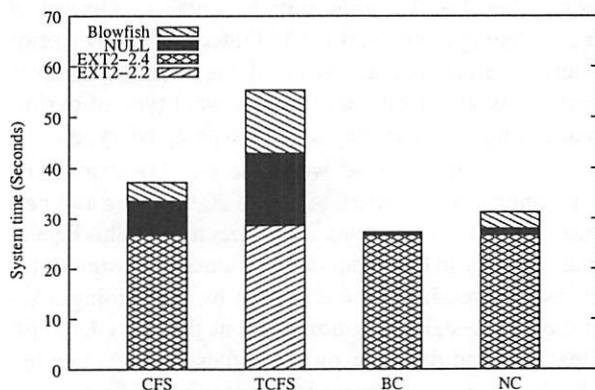


Figure 3: Am-utils build system time. Each bar represents an EXT2 configuration, a NULL configuration, and an encrypting configuration.

The system time used by a process demonstrates how much CPU was used by the additional file system overhead and encryption. These results can be seen in Figure 3 and the last two rows of Table 2. CFS has a user-space process, `cfsd`, which performs all encryption. BestCrypt has a kernel-space thread that performs encryption. We added the time used by these processes into the system time to represent the total time used on behalf of the process. TCFS makes use of `knfsd`, but as this can be on a remote server we do not include its overhead. These results show that the overhead for CFS and TCFS is quite large compared to BestCrypt and NCryptfs. This was expected because of the added network overhead. BestCrypt has a simpler interface and hence a smaller overhead than NCryptfs.

The user times for all tests were within 2% of the EXT2 configuration. This is expected as the encryption happens outside of the process in all of these systems.

5.2.4 Postmark Results

The elapsed time overheads for Postmark are shown in Figure 4 and the first two rows of Table 3. This shows that for I/O-intensive operations, all of the cryptographic file systems we tested have a non-negligible impact on system performance. Increasing security often affects performance, so these results were consistent with our expectations. The results for CFS and TCFS show a larger performance overhead than for Am-utils, but the overhead of user-space data copies and the NFS protocol explain this. BestCrypt performed significantly better in the Am-utils test than in the Postmark test; this is because BestCrypt uses a single kernel thread for encryption that often disables interrupts. The I/O intensive nature of the Postmark test exposes this behavior. Other elapsed times did not significantly increase when encryption was used.

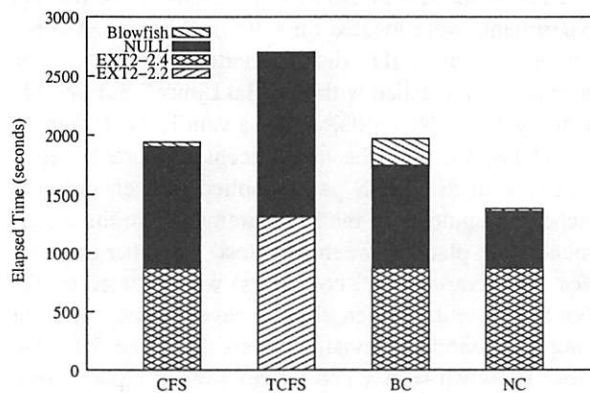


Figure 4: Postmark elapsed time. Each bar represents an EXT2 configuration, a NULL configuration, and an encrypting configuration.

Configuration	CFS	TCFS	BC	NC
Elapsed Time – NULL	119	106	101	56
Elapsed Time – BF	123	106	127	59
System Time – NULL	553	50	95	51
System Time – BF	821	118	280	156

Table 3: Postmark percentage overheads over EXT2

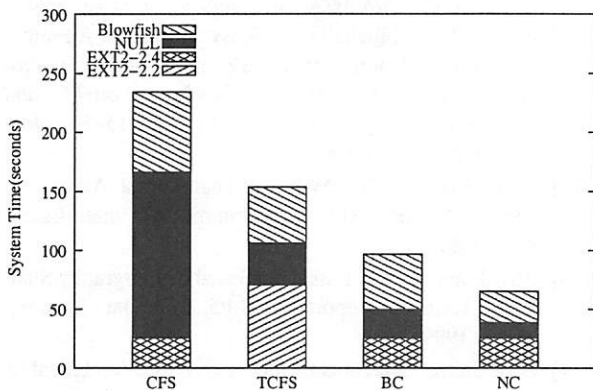


Figure 5: Postmark system time. Each bar represents an EXT2 configuration, a NULL configuration, and an encrypting configuration.

Figure 5 and the last two rows of Table 3 shows system time in the same manner as described in Section 5.2.3. CFS has the worst performance degradation over EXT2, caused by the excessive number of data copies from user-space to kernel-space and within the network stack. TCFS also suffers from data-copy overheads in the network stack. BestCrypt and NCryptfs both have smaller overheads than CFS and TCFS.

6 Conclusions

Our main contribution is in designing and building a cryptographic file system that for the first time, to our best knowledge, was developed with the express goal of balancing all of these four conflicting aspects: security, performance, convenience, and portability.

We achieved high security by including support for many ciphers and authentication methods, addressing vulnerabilities in OS caches and the task manager, separate OS name spaces, separate encryption from authentication keys, session-based and process-based encryption, reduced user need for superuser privileges, and key timeouts that are transparent to processes.

We achieved high performance by designing NCryptfs to run in the kernel. Our performance benchmarks show a small 5% overhead for normal system operation.

We achieved ease of use by allowing administrators and users alike to tailor the levels of security, performance, and convenience to their needs: by providing

encryption and authentication that is transparent to users and processes; by allowing users to quickly attach and detach from NCryptfs; by supporting ad-hoc encryption groups for shared yet secure collaboration; and by automatic loading of ciphers and authentication modules.

Lastly, we achieved high portability by building NCryptfs as a stackable file system. This allows users to use any file system as the backing store for encrypted data, and helps to reduce the development and porting effort of NCryptfs to other systems. Although our first prototype works in Linux alone, we developed NCryptfs using the FiST stackable templates system, which also supports Solaris and FreeBSD [34].

6.1 Future Work

In the immediate future, we plan to integrate a lockbox mode and cryptographic checksumming into NCryptfs. One possible method to achieve integrity assurance is to store block-by-block checksums of the file in a separate checksum file. This technique is similar to using index files in size-changing file systems [31]. We will also investigate the best way to store a unique IV along with each file, so that NCryptfs does not rely on the inode number of files.

Currently, a stackable file system can change lower-level file system data independently from upper-level data. Data on multiple levels must be associated within OS caches to present a coherent system view. We plan to modify the VFS caches to associate each upper level cache object (dentry, inode, and page) with its lower level object [7].

Presently, NCryptfs exposes the owner and other inode meta-data of a file. We cannot remove all of this structure without making incompatible changes to the lower-level file system. We can perturb some data, such as adding padding to the file's name or storing ownership information outside of the lower-level inode. These operations may complicate backup and restore. We also expect this to decrease system performance.

7 Acknowledgments

We wish to thank Jeffrey Osborn, Amit Purohit, and Kiran Reddy for reviewing early drafts of the paper and assisting with benchmarking. We thank Tzi-cker Chiueh, Fred B. Schneider, Jukka T. Virtanen, our shepherd Bennet Yee, and the anonymous USENIX reviewers for the valuable feedback they provided. This work was partially made possible by an NSF CAREER award EIA-0133589, and HP/Intel gifts numbers 87128 and 88415.1.

References

- [1] M. Blaze. A cryptographic file system for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*, 1993.
- [2] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 245–252, June 2001.
- [3] M. Corner and B. D. Noble. Zero-interaction authentication. In *The Eighth ACM Conference on Mobile Computing and Networking*, September 2002.
- [4] R. Dowdeswell and J. Ioannidis. The CryptoGraphic Disk Driver. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, June 2003.
- [5] A. Grünbacher. Extended attributes and access control lists. <http://acl.bestbits.at>, 2002.
- [6] P. C. Gutmann. Secure filesystem (sfs) for dos/windows. www.cs.auckland.ac.nz/~pgut001/sfs/index.html, 1994.
- [7] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. In *Proceedings of Fifteenth ACM Symposium on Operating Systems Principles*. ACM SIGOPS, 1995.
- [8] J. Hennessey. The Future of Systems Research. *IEEE Computer*, 32(8):27–32, 1999.
- [9] IEEE/ANSI. Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment: Protection, Audit, and Control Interfaces [C Language]. Technical Report STD-1003.1e draft standard 17, ISO/IEC, October 1997. *Draft was withdrawn in 1997*.
- [10] Jetico, Inc. BestCrypt software home page. www.jetico.com, 2002.
- [11] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance. www.netapp.com/tech/library/3022.html.
- [12] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Technical Report RFC 2104, Internet Activities Board, February 1997.
- [13] A. D. McDonald and M. G. Kuhn. StegFS: A Steganographic File System for Linux. In *Information Hiding*, pages 462–477, 1999.
- [14] Microsoft Corporation. Encrypting File System for Windows 2000. Technical report, July 1999. www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp.
- [15] R. Nagar. *Windows NT File System Internals: A developer's Guide*, pages 615–67. O'Reilly, September 1997. Section: Filter Drivers.
- [16] National Association of Insurance Commissioners. Graham-Leach-Bliley Act, 1999. www.naic.org/GLBA.
- [17] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.0.4 edition, February 2000. www.am-utils.org.
- [18] R. Pike. Systems Software Research is Irrelevant. Bell Labs, February 2000. www.cs.bell-labs.com/who/rob/utah2000.pdf.
- [19] R. Power. Computer Crime and Security Survey. *Computer Security Institute*, VIII(1):1–24, 2002. www.gocsi.com/press/20020407.html.
- [20] N. Provos. Encrypting virtual memory. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.
- [21] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 15–30, Monterey, CA, January 2002.
- [22] R. L. Rivest. The MD5 Message-Digest Algorithm. Technical Report RFC 1321, Internet Activities Board, April 1992.
- [23] RSA Laboratories. Password-Based Cryptography Standard. Technical Report PKCS #5, RSA Data Security, March 1999.
- [24] J. H. Saltzer. Hazards of file encryption. Technical report, 1981. <http://web.mit.edu/afs/athena.mit.edu/user/other/a/Saltzer/www/publications/csrrfc208.html>.
- [25] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 2 edition, October 1995.
- [26] U.S. Dept. of Health & Human Services. The Health Insurance Portability and Accountability Act (HIPAA), 1996. www.cms.gov/hipaa.
- [27] VERITAS Software. Veritas file server edition performance brief: A postmark 1.11 benchmark comparison. Technical report. <http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf>.
- [28] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the Eighth Usenix Security Symposium*, August 1999.
- [29] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
- [30] E. Zadok. Stackable file systems as a security tool. Technical Report CUCS-036-99, Computer Science Department, Columbia University, December 1999.
- [31] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 289–304, June 2001.
- [32] E. Zadok, I. Bădulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.
- [33] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, June 1999.
- [34] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.

A Binary Rewriting Defense against Stack based Buffer Overflow Attacks

Manish Prasad and Tzi-cker Chiueh
SUNY Stony Brook

{mprasad, chiueh}@cs.sunysb.edu

Abstract

Buffer overflow attack is the most common and arguably the most dangerous attack method used in Internet security breach incidents reported in the public literature. Various solutions have been developed to address the buffer overflow vulnerability problem in both research and commercial communities. Almost all the solutions that provide adequate protection against buffer overflow attacks are implemented as compiler extensions and hence require the source code of the programs being protected to be available so that they can be re-compiled. While this requirement is reasonable in many cases, there are scenarios in which it is not feasible, e.g., legacy applications that are purchased from an outside vendor. The work reported in this paper explores application of static binary translation to protect Internet software from buffer overflow attacks. Specifically, we use a binary rewriting approach to augment existing Win32/Intel Portable Executable (PE) binary programs with a return address defense (RAD) mechanism [1], which protects the integrity of the return address on the stack with a redundant copy. This paper presents the disassembly and instrumentation issues involved in static binary translation, how our tool achieves satisfactory disassembly precision in the presence of indirect branches, position-independent code sequences, hand crafted assembly code and arbitrary code/data mixing, and how it ensures safe binary instrumentation in most practical cases. The paper reports our experiences with this approach, based on results of applying the resulting prototype to rewriting several commercial grade Windows applications (Ftp server, Telnet Server, DNS server, DHCP server, Outlook Express, MS FrontPage, MS Publisher, Telnet, Ftp, Winhlp, Notepad, CL compiler, MS NetMeeting, MS PowerPoint, MS Access, etc.), as well as experimentation with published buffer overflow exploits.

1 Introduction

Buffer overflow attacks exploit a particular type of program weakness: lack of array/buffer bound check in the compiler or in the applications. Accordingly, the ideal solution to the buffer overflow vulnerability problem is to build a bound checking mechanism into the compiler, or to require applications to strictly follow a program-

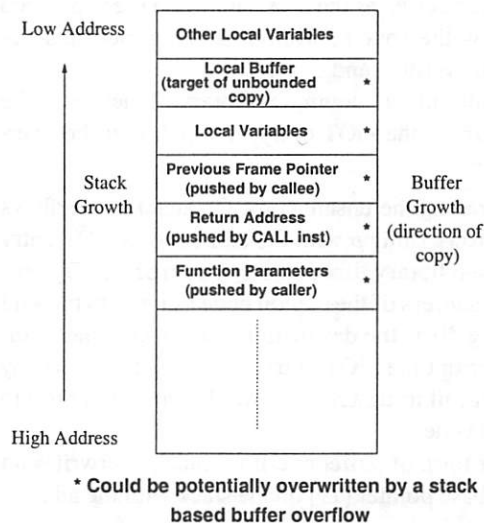


Figure 1: The typical stack layout of a function when it is called, and how some of the stack entries, including the return address could be corrupted by an unsafe copy operation.

ming guideline that checks the bound of an array/buffer upon each access. Neither solution is considered practical at this point. A more promising approach is to transform a given application into a form that is immune from buffer overflow attack without requiring any modification to the compiler or the application itself.

To understand a buffer overflow attack, consider a typical stack layout when a function is called, as shown in Figure 1. Lack of bound checking during a buffer copy operation causes areas adjacent to the buffer (as shown by * in Figure 1) to be overwritten. A generic buffer overflow attack [2] involves exploiting such an unsafe copy to overwrite the return address on the stack with the address of a piece of malicious code, which is injected by the attacker and most likely reside also on the stack; when the RET instruction (which pops the return address from the stack) in the victim function is executed, program control is transferred to the injected malicious code.

A less common form of buffer overflow attack involves corrupting memory pointer variables on the stack instead of return addresses [18]. The requirements for such an attack to occur are:

1. A pointer variable *p* that is physically located on the stack after the buffer *a[]*, to be overflowed,
2. An overflow bug that allows overwriting this pointer *p* by overflowing *a[]* (taking user-specified data as source), usually with the address of a Global Offset Table (GOT) entry, which contains the address of a dynamic library function,
3. A copy function such as `str(n)cpy/memcpy` which takes **p* as the destination and user-specified data as the source, without *p* being modified between overflow and copy,
4. A call to a common library function (like `printf`), the GOT entry of which is to be overwritten.

So leveraging the unsafe copy, the attacker overflows *a[]*, thus overwriting *p* with the address of a GOT entry of a common library function, say `printf()`. By providing the address of the exploit code as input to the safe copy taking **p* as the destination, the attacker has managed to corrupt the GOT entry of `printf()`. So any subsequent call to `printf()` would transfer control to the exploit code.

Another form of buffer overflow attack overwrites up to the old base pointer [19] on the stack with the address of malicious code, so that when the caller function returns, the control is transferred to the exploit code.

A seemingly non-stop stream of buffer overflow attacks [3, 4] has called for an effective and practical solution to protect application programs against such attacks on the Windows platform. Past approaches to protecting programs against buffer overflow vulnerabilities relied on compiler extensions, either to perform array bounds check or to prevent the return address on the stack from being overwritten. Although fairly successful in preventing most conventional buffer overflow attacks [2], these approaches require access to program source code. All known systems that take such an approach require the availability of the protected application's source code. While integrating software protection into a compiler is technically desirable, this approach exhibits several practical limitations. First, requiring access to source code makes it difficult to protect third-party legacy applications whose source code is unavailable for various reasons. Second, because modern software applications tend to be built on third-party libraries, access to the source code of these libraries again is unlikely. Finally, for those program segments that are written in assembly code directly, their high-level-language source code that is amenable to compiler analysis simply does not exist. The goal of this paper to describe our experiences and the extent of success that we have achieved in applying a combination of well known disassembly techniques to implement a binary rewriting solution that aims to provide the same level of protection

as its compiler-based counterpart

There are two major technical challenges in applying binary rewriting to the buffer overflow attack problem. First, to determine where to insert protection instructions, the boundary of each function in an input program needs to be clearly identified, which in turns requires an accurate disassembler that can correctly decode each instruction in an executable binary. Unfortunately, 100% disassembly accuracy is difficult because the problem of distinguishing code from data embedded into code regions is fundamentally undecidable. Second, even if the function boundaries are successfully identified, inserting protection code into a given binary without disturbing the addresses used in its existing instructions is itself non-trivial. The main problem here is that in many cases it is possible that the binary does not have enough spare space to hold a jump instruction to the inserted protection code, let alone to hold the protection code itself.

Section 2 surveys related work in the areas of static binary translation and buffer overflow defense. Section 3 discusses the design and implementation of our disassembly engine and the binary instrumentation issues with emphasis on the approaches we employ to ensure program safety and to preserve the semantics of input programs. Section 4 details the software architecture and implementation of the binary rewriting RAD prototype. Section 5 presents experimental results on the prototype's resistance to attacks, ability to preserve the semantics of applications, space cost and performance overheads. Finally, section 6 summarizes the main results of this work and charts out directions for future research.

2 Related Work

Among past efforts in binary rewriting, ATOM [5] and EEL [6] run on RISC architectures, where the disassembly problem is simplified due to uniform instruction size. Etch [7] is a tool for rewriting Win32/Intel PE executables primarily for optimization. LEEL [8] works on Linux/x86 binaries, albeit with limitations with respect to control flow analysis in presence of indirect control transfer instructions and arbitrary code/data mixing. UQBT [9] is an architecture independent static binary translation framework for migrating legacy applications across processor architectures. Galen Hunt's Detours [24], is a system for run-time binary interception of Win32 functions.

Most buffer overflow defense proposals involve compile-time analysis and transformation. Stackguard [10] and Microsoft Compiler Extension [11] place 'canary words' on the stack between the local variables and return address at the function prologue, and monitors the return address on the stack by checking the integrity of the 'canary word', at the epilogue. Both are vulnerable

to attacks based on corrupting old frame pointers [19] on the stack or local pointer variables [18]. Stackshield [12] and RAD [1] save a copy of the return address at the prologue and compare it with the return address on the stack at the epilogue. Our binary rewriting implementation is based on this model of buffer overflow defense. Both are resilient to frame pointer based attacks [19] but vulnerable to memory pointer corruption [18] attacks. IBM's gcc extension [14] also does local variable reordering, placing pointer variables at lower addresses than buffers in addition to the above, offering some protection against memory pointer attacks [18], unless the unsafe copy is from higher to lower indices of the array. CASH [15] and others [16] perform array bounds check to prevent overflow of buffers. CASH achieves significant overhead reduction (4% overhead) by exploiting Intel segmentation hardware as compared to the others in this category, which typically incur very high overhead (70% to 140%).

Other proposed approaches to protect programs from buffer overflow attacks rely on run-time interception and checking. Lucent Bell Labs' Libsafe [20] intercepts unsafe library calls at run-time and performs bounds checking on the arguments e.g. for strcpy(), it would check the length of the source string and check it against the upper bound on the length of the destination string based on the current frame pointer value. Although it prevents the return address from being modified, it is possible to corrupt local pointer variables. Libverify [20] performs dynamic binary translation to perform return address check. However, we suspect that Libverify might incur high overhead, since it adds checking code and performs code instrumentation at run-time. A very recent example of applying optimized run-time interpretation to security problems is program shepherding [21], which is built on top of a dynamic optimization framework called RIO. Apart from offering advantages like complete transparency, it achieves significant overhead reduction as compared to what one would expect from an interpretation/emulation system using a variety of optimization techniques viz. traces and interpreted code caching. Another example of dynamic binary translation (applied to parallel computing) is Paradyn [27]. Jun Xu et. al [28] suggest a hardware-based solution against buffer overflow attacks without requiring access to program source code.

To the best of our knowledge, the binary-rewriting RAD system described in this paper is the first attempt that employs static binary translation to protect existing binaries against buffer overflow attacks without requiring access to program source code, symbol tables or relocation information.

3 Binary-Rewriting Return Address Defense

A successful binary rewriting RAD system requires identifying the boundary of every procedure in the input program and inserting a protection instruction sequence into every procedure without disturbing the input binary's internal referencing structure. The following two subsections discuss in more detail these two issues and their associated solutions.

3.1 Binary Disassembly

3.1.1 Disassembly Challenges

To accurately locate the procedure boundary, one needs to identify each instruction in the binary through a disassembler. There are two main classes of disassembly algorithms [22]. A *linear sweep* algorithm starts with the first byte in the code section and proceeds by decoding each byte, including any intermediate data byte, as code, until an illegal instruction is encountered. A *recursive traversal* algorithm starts at the program's main entry point and proceeds by following each branch instruction encountered in a depth-first or breadth-first manner, essentially a control flow analysis. Neither approach is 100% precise. The chief impediments to accurate disassembly are:

1. Data embedded in the code regions,
2. Variable instruction size,
3. Indirect branch instructions,
4. Functions without explicit CALL sites within the executable's code segment,
5. Position independent code (PIC) sequences, and
6. Hand crafted assembly code.

1) and 2) render the linear sweep algorithm less effective than ideal, whereas 3), 4) and 5) degrade the efficacy of the control flow analysis used in the recursive traversal algorithm.

Distinguishing code from data in a binary file is a fundamentally undecidable problem. Because the linear sweep algorithm decodes each byte as code as long as it looks like a legitimate code byte, it ends up interpreting many data bytes as instructions. The reason for this behavior is that, in the Intel x86 instruction set, 248 out of 256 possibilities can be a legitimate starting byte for an instruction, making it more likely to mistake data for instruction. The fact that the Intel x86 instruction set allows variable instruction size further aggravates the problem of code/data distinction. Consider the following example sequence of bytes:

```
0x0F 0x85 0xC0 0x0F 0x85 . . . . .
```

If we consider 0x0F as a code byte then we'll end up with the following disassembly:

```
jne offset
```


On the other hand if we consider 0x0F as a data byte and 0x85 as a code byte, then we get something like:

```
0x0F    // data
test eax, eax
jne offset
```

Thus a single disassembly error could result in many subsequent bytes being interpreted incorrectly, with the extent of error potentially unbounded. In contrast, a fixed-instruction-size architecture exhibits a self-correcting property: an interpretation error for one instruction word does not propagate to the next instruction word.

The recursive traversal algorithms cannot obtain 100% accurate disassembly results, either, because it is difficult to construct a complete control flow of an input binary in the presence of indirect branch instructions such as `call/jmp reg32` (e.g. `call eax`) or `call/jmp m32` (e.g. `jmp dword[esp + xx]`). One solution to this problem is to perform additional data flow analysis such as inter-procedural slicing and/or constant propagation [23] to figure out at compile time the value of the register or memory location used in indirect control transfer instructions. Apart from being difficult to implement, such an approach tends to greatly increase the disassembly time and itself does not guarantee 100% accuracy.

Procedures for which no explicit call sites in the input program can be identified include exception or signal handlers, callback functions, which is rife in GUI applications, and procedures all calls to which are through indirect branch instructions. Because there is no identifiable call to these functions, they cannot be discovered through control flow analysis, and as a result may be misclassified as data. In practice, signal/exception handlers pose few problems because their entry points are included in the program header in some cases.

The addressing in position independent code (PIC) does not rely on any particular position in the program's address space. Thus PIC code and jump table never have absolute address references. Instead the references are in the form of offsets with respect to a base value that is known at run time, mostly through the `eip` value. For example,

```
10: call 10
15:
:
25: pop eax    // gets the return addr
           // value 15 into eax
26: call dword[eax + 20] // call foo
:
:
35: // foo
```

In this case, in spite of having explicit `CALL` sites, standard control flow analysis cannot discover the target location of the function `foo()`.

Hand crafted assembly code makes it difficult to identify procedure boundaries because they do not necessarily follow the code conventions established by standard compilers. These conventions provide useful hints to resolve potential ambiguities. As an example of code convention violation, some assembly code programs jump from one function into another function without going through the latter's main entry point.

3.1.2 Disassembly Engine Implementation

Our disassembly engine is built on the x86 instruction set parsing and disassembly capabilities of an existing disassembler [13]. We use a combination of well-known disassembly techniques, viz. recursive traversal and linear sweep (described briefly in the previous subsection 3.1.1) and complement them with compiler-independent pattern matching heuristics. We assume that the data expected in a code section are typically dispatch tables (address bytes), strings and compiler alignment bytes. Since the goal of this project is to insert protection code into every procedure of the input binary, we should identify as many code bytes as possible; otherwise the transformed binary may have security holes. However, we place maintenance of original program semantics at a higher priority than security, so whenever in doubt we mark bytes as data instead of code, thus avoiding unsafe binary instrumentation. The following is a step-by-step description of the disassembly process:

1. Identify potential address bytes for dispatch table discovery and strings. Dispatch tables typically contain code section addresses. Since we know the address range of the code section, we can mark any sequence of 4 bytes, which have a value that lies within this address range as 'potential address' bytes. Sequences of printable characters that have a certain minimum length and are terminated by a null character are marked as 'potential strings.'
2. Starting from the program's main entry point, which is obtained from the input binary's PE header [17], we perform a control flow analysis on the binary to traverse the paths of the program's control flow graph. All code bytes identified in this step are marked as 'definitely code' and all associated data bytes marked as 'definitely data'. We also identify targets of `CALL` instructions as function entry points and targets of conditional and unconditional jump instructions as jump targets. Since this step can distinguish data from code with 100% accuracy, it overrides analysis results from other steps whenever there is a conflict. For example, the following byte sequence will be identified as a `call` instruction because the result from Step (2) over-

rides that of Step (1).

```
Identified as instruction
'call dword[0x30001344]' in Step (2)
<----->
0xFF 0x15 0x44 0x13 0x00 0x30
<----->
Identified as 'potential
address' (0x30001344) in
Step (1)
```

3. To identify the entry points of potential callback functions, for which there are no explicit call sites, we look for instruction sequences such as:

```
push imm32
mov reg32, imm32
```

Typically the target address of a callback function is usually passed as an argument to some function, with which the callback function is registered. Such an argument could be passed through the stack as an immediate value (`push imm32`) or through a register, which contains the address value (`mov reg32, imm32; push reg32`). If the byte at `imm32` has not been identified as a 'potential address' or a 'potential string' in Step (1), and if it looks like a legitimate instruction starting byte, we consider it as a function entry point (although despite being a legitimate code byte it may not actually be a function entry point) and proceed disassembling the subsequent bytes as instructions.

4. To identify other types of functions for which there are no explicit call sites, we next look for bytes in the code section that have not been identified as code or data yet. Every time such a byte is located, we start instruction parsing if it looks like a legitimate instruction starting byte. In both Steps (3) and (4), the point where such instruction parsing begins is called a 'reset point'. Instruction parsing continues until an unconditional branch instruction (`ret` or `jmp`) is encountered. If the result of an instruction parsing procedure is inconsistent with any previously identified byte or leads to an illegal instruction byte, the result since the reset point is revoked and all the bytes from the 'reset point' to the current position are marked as data, thus avoiding any potentially unsafe binary rewriting. After an unconditional branch we look for the next suitable 'reset point' to start the next instruction parsing attempt.
5. Because any sequence of instruction bytes should end with an unconditional branch instruction (`jmp` or `ret`), we look for code sequences that end without an unconditional branch (`jmp` or `ret`) instruction and mark such code sequences as data. This check provides a final line of defense to eliminate any potentially incorrectly identified instruction se-

quences. For example, in the following code sequence, Bytes 1 to 3 will be marked as data.

```
1: mov eax, ebx
3: push eax
4: data
```

Also, the byte next to an unconditional branch has to be either a data byte or if it is a code byte, it must be a branch target (as the previous instruction, being an unconditional branch, doesn't fall through). Therefore, in the case that the byte next to an unconditional branch is a code byte and has not been marked as a function entry point or a jump target, we mark it as a function entry point, even though they could just as well be targets of a branch instruction inside the same function.

The motivation behind this "optimistic" identification of functions, as seen in steps 3) and 5) will be explained in subsection 4.3.

3.2 Binary Instrumentation

Because it is not always possible to derive the high-level control flow of an input binary, the process of inserting additional code to counter buffer overflow attacks must proceed in a way that does not disturb the memory references used in the instructions of the binary program that is to be protected.

3.2.1 Where to Insert RAD Code

The additional code required by RAD [1] involves

- Saving a copy of the return address on the stack in the return address repository (RAR) at the function prologue, and
- Checking the return address on the stack with the saved copy in the RAR at the function epilogue, popping it off the RAR in the event we have a match, or flag an exception otherwise.

Instead of adding function prologues and epilogues to every function, we choose to do so only for 'interesting' functions, which are functions that contain a sequence of instructions for stack frame allocation and deallocation for local variables. A function without local variables could never be vulnerable to a stack based buffer overflow.

3.2.2 How to Insert RAD Code

So as to not disturb the original binary's address space, we choose to create a separate new code section, not present in the original PE binary (information regarding the PE format is in [25]), appended to the end of the original binary to hold the additional prologue and epilogue code for each function. Moreover this new section, mapped to a non-interfering portion of the address space, will be set as read-only. Thus neither the RAD

code is corrupted by the application nor is the application corrupted by the RAD code. To redirect control to the inserted code at a function's prologue and epilogue, we need to replace some instructions at the function prologue and epilogue with a JMP to the corresponding RAD code. When such an instrumented function is invoked, the JMP instruction, which replaces the prologue, transfers control first to the RAD prologue code, then executes the original prologue instructions and then jumps back to the original function to continue execution from the instruction immediately after the original function prologue. Epilogue instructions are replaced in a similar manner. However, the execution proceeds first with a JMP to the epilogue code in the new section, first executing the original epilogue instructions until the RET, then the RAD epilogue checking code and then return if there are no problems. Because the size of an unconditional JMP instruction is 5 bytes, we need at least 5 bytes worth of instruction space to accommodate a JMP instruction. Instructions that are target of existing branch instructions cannot be replaced.

A function prologue, which needs to allocate stack space for local variables, typically comprises 3 instructions :

```

1.  push ebp // save old frame ptr
    // (1 byte instruction)
2.  mov ebp, esp // set the top of
    // the stack as the
    // current frame ptr
    // (2 byte inst)
3.  sub esp, x // allocate x bytes on
    // the stack for local
    // variables (3 to 6
    // byte instruction)

    or
    add esp, -x

```

Alternatively it could also be done using the ENTER instruction, however most compilers do not use ENTER for stack frame allocation. Thus, an 'interesting' function prologue includes at least 6 bytes worth of instructions. Hence, we can comfortably instrument an 'interesting' function prologue to redirect control to the RAD prologue code using a 5-byte JMP instruction. On the other hand, a typical stack frame deallocation instruction sequence looks like one of the following three cases:

```

1.  add esp, x // dealloc. stack
    // space, x bytes
    // were allocated
    // (3-6 byte inst)
    pop ebp // restore caller's
    // frame ptr (1 byte)
    ret // return (1 byte)
2.  mov esp, ebp // dealloc. stack
    // space, any
    // number of bytes

```

```

// allocated on the
// stack (2 byte
// instruction)
pop ebp // restore caller's
// frame ptr (1
// byte)
ret // return (1 byte)
3.  leave // dealloc. stack
    // frame & restores
    // old frame ptr
    // (1 byte)
    ret // return (1 byte)

```

From 2) and 3), we see that stack frame deallocation could be done with 2 to 4 bytes worth of instructions. So we need to replace some more instructions in addition to the stack frame deallocation instructions to hold a JMP instruction. In most cases, we do find enough space this way. However, it is possible that the first instruction of the stack frame deallocation sequence is a jump target, e.g.:

```

jne x
:
x: leave
ret

```

In this case, if we replace instructions prior to LEAVE, then the jump target x would be disturbed. From our experiences, the scenario of not being able to find 5 bytes worth of instructions at a function's epilogue does occur in practice but is relatively rare. For such a situation to occur in practice, two conditions need to be met:

- a) Most development environments on Windows, by default, set certain compilation options which generate calls to stack checking code, prior to stack frame deallocation, to check for adherence to certain calling conventions (which basically dictate caller and callee duties as regards function frame initialization and cleanup). Calling convention adherence check is desirable because of functions being called using function pointers and calls to library functions. If we disable these options the compiler won't generate these stack checking calls and thus will not generate extra bytes prior to stack frame deallocation.
- b) There should be a high level code sequence like:

```

goto label;
:
:
label:
return;

```

So in such rare scenarios (our experiments show typically 0.03 - 3% of all functions, sec. 5.2.2, table 9), we use a simple although expensive approach to solve this problem. When not enough instructions are available, we replace the first byte of the instruction prior to ret

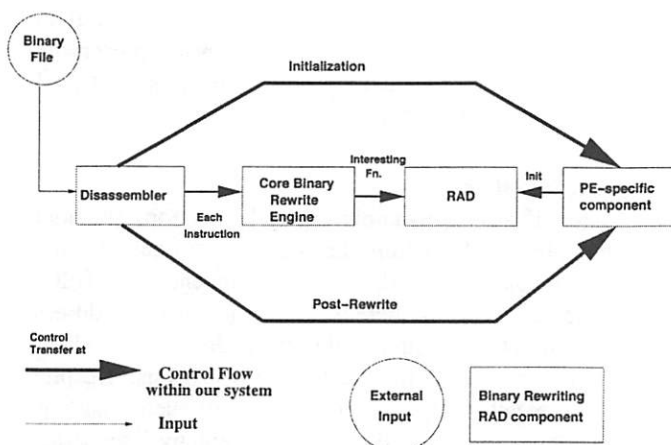


Figure 2: The software architecture of the binary-rewriting RAD prototype, which consists of a disassembler, a core binary rewriting engine, a RAD component, and a PE component.

with an `int 3` (breakpoint interrupt) instruction, which corresponds to a software interrupt, and install a corresponding exception handler. When an `int 3` instruction is executed, it generates a Debugger Breakpoint Exception, and the handler gains control to perform return address check. Because this exception handler is executing the user space, control transfer to our handler is similar to an intra-privilege level far call, which means that there is no stack switching and the exception handler can access the return address on the stack. For details regarding how the stack evolves during the execution of a software interrupt handler, please refer to [17]. The reason why we chose the debugger breakpoint exception is that this exception is not used normally unless the program is being debugged. However, while being debugged under a debugger, the control is transferred to the debugger when an `int 3` instruction is executed, and our exception handler will not be executed.

4 Prototype Implementation

4.1 Software Architecture

The binary-rewriting RAD tool comprises the following logical components: a disassembler, a core binary rewrite engine, a RAD component, and a PE (Portable Executable format specific) component. The disassembler functions in two main phases. In the first phase, it performs code/data and branch target identification covering all bytes in the code section, and in the second, it outputs the assembly instructions starting from the first byte in the code section. The core binary rewrite engine, independent of the binary format, hooks into the disassembler in the second phase to gain control at every instruction processed to look for 'interesting' function prologues and epilogues to instrument. This com-

ponent handles all the issues involved in adding instrumentation code outlined in the section 3.2. Since the instructions that make up an 'interesting' pattern need not be contiguous, this component maintains a window of five instructions (current instruction and four previous ones), which is flushed whenever a branch target is encountered, so that we don't run over any jump target. The engine attempts to identify 'interesting' patterns in the window every time a new instruction is added. All the RAD code and its associated data are added to a new section at the end of the input binary. The RAD component implements the Return Address Defense mechanism. The prologue stub saves the return address on the stack to the Return Address Repository (RAR) and the epilogue stub keeps popping the RAR stack till it finds the return address currently on the top of the stack or till the RAR is empty in which case it flags an exception. This repeated popping ensures that the return from any of the caller's ancestors (from the current call stack), does not generate false security alarms. This scenario occurs in case of `setjmp()/longjmp()` as well as compiler optimizations which cause functions to return straight to the caller's caller if the first return is to the caller's return section (e.g. tail recursion). The PE component initializes the binary rewrite process by adding a new section header in the section header table and setting up its fields appropriately; it also aligns the new section (called the `.RAD` section) that holds the RAD code, depending on where it should be loaded at run time (page boundary next to the end of the previous section) and where it should be stored in the binary file (file alignment boundary after the end of the previous section).

4.2 RAR Initialization

The `.RAD` section is set as read/write/executable. It needs to be writable since the RAR is also a part of this section in addition to the RAD code. However, at run time the non-RAR part of this section needs to be set to read-only, through a Win32 API call `VirtualProtect()`. This is to create mine-zones on both the sides of the RAR to prevent attackers from overflowing the RAR. The key issue here is how to locate the entry point of `VirtualProtect()`. There are several cases to consider. First, it is possible that the input program also uses `VirtualProtect()` for some other purpose and thus the pointer to its entry point can be located by scanning the binary statically. Otherwise, if the input program does not need `VirtualProtect()`, the PE component needs to add code to locate its entry point at run-time.

For the first case, two PE headers of interest here are the Import Address Table (IAT) and the Import Name Table (INT). The two tables can be reached from the import directory, whose location in turn is obtained from the `DataDirectory` array in the PE `OptionalHeader`. Each

entry in the IAT, at run time, contains the address of an imported function and the on-disk binary file. For each IAT entry there is a corresponding entry in the INT, which points to the name of the corresponding imported function. At load time, the loader overwrites the IAT entries with the virtual addresses where the corresponding functions are mapped. To locate the entry point of `VirtualProtect()`, we look up the INT by its name and retrieve the index of the associated IAT entry if there is a match. If `VirtualProtect()` is already imported by the input program, then its entry point is readily available from its IAT entry. If there is a match in the INT but the corresponding IAT entry is empty, then we need to dynamically resolve the entry point of `VirtualProtect()` by calling `GetModuleHandle()`, which gets the base address of the DLL containing the API function in question, and then calling `GetProcAddress()`, which gets the address of the desired API function from the export directory of its containing DLL. The entry points of `GetModuleHandle()` and `GetProcAddress()` themselves are obtained from the IAT through their containing DLL, `kernel32.dll`. In the case that none of these API functions are themselves imported, which is quite rare, then the PE component needs to emulate the operation of `GetModuleHandle()` and `GetProcAddress()`. This emulation idea is derived from an undocumented virus code and is based on the following observation: At the program entry point, the top of the stack contains a return address, which points to somewhere inside the function `CreateProcess()`, which in turn belongs to the DLL `kernel32.dll`. With this address, one can scan through the memory until where `kernel32.dll` is mapped is found. Once the base address of `kernel32.dll` is known, the entry points of `GetModuleHandle()` and `GetProcAddress()` are available through the DLL's export table, and in turn the entry point of any API function can be identified through these two functions.

Finally, the entry point of the executable in the PE header should be changed to the new initialization code, which locates the entry point of `VirtualProtect()`, and calls it to protect two pages surrounding the RAR.

4.3 Limitations

4.3.1 Security Weaknesses Due to Disassembly Limitations

Two aspects of disassembly that relate to sources of false security alarms and security loopholes are:

- Functions Missed by our Disassembly Engine (False Negatives)
- Falsely Identified Functions (False Positives)

Let's look at each of these aspects and evaluate when and how these could result in false alarms or missed attacks. There is a trade-off between security and

program correctness. While we make every effort to seal all known security holes, we consider preserving original program semantics as a more important goal than attempting perfect security.

False Negatives

These (if any) are mainly callback functions (without an explicit `CALL` within the code section) and/or functions invoked using Position-Independent Code (PIC) sequences (wherein there would be no absolute address references to a function within the code section). These are not covered by pure control flow analysis. Despite this, typically, only a certain fraction of such functions get missed out. The following "representative" scenarios should make this clear. Functions missed by the control flow analysis step could be:

- a) partly/fully misidentified as data
- b) identified fully as code

If the start or the end of a function is misidentified as data, then we might miss out on an interesting prologue or epilogue respectively. Either cases result in an unprotected return, which might turn out to be a security loophole, if that particular function has a buffer overflow vulnerability. Ditto is the case when a function is fully identified as data. If a piece of code somewhere in the middle of a function is misidentified as data, then the function is misidentified as data, then the function is divided into two, and hence all returns in this function beyond this dividing point would be treated as a part of an uninteresting function, and hence are left uninstrumented and could miss an attack.

A function with its body fully identified as code, could still be missed out during control flow analysis and have their unidentified entry points preceded either by data or an unconditional branch instruction from the previous function. In either of the cases, we would indeed mark the function entry point (last step (5) of disassembly engine sec. 3.1.2). When data preceding the entry point of such a function aligns properly with the code bytes to form a legitimate instruction sequence, an originally interesting prologue could become uninteresting, thus exposing an attack opportunity. In all the cases presented so far, however, program semantics are not jeopardized. But if data misidentified as code turns an uninteresting function prologue to an interesting one, it might generate a false alarm, if the epilogue happens to be interesting. Another false alarm scenario is if the function entry point is preceded by some data and the first identified code byte happens to be a jump target (happens with inter-procedural jumps), in which case the two functions get merged into one. However, inter-procedural jumps occur only in handcrafted assembly or as in `setjmp()/longjmp()` cases.

Apart from functions, jump targets reached by PIC jump tables could be missed. This could affect program correctness, if these targets happen to be within instrumented prologues or epilogues, a very unlikely scenario, though.

False Positives

Functions with multiple entry points are treated as two separate functions. Targets of PIC jump tables, which cannot be discovered statically could get marked as function entry points, if they lie immediately after an unconditional branch or a sequence of data bytes (last step (5) of disassembly engine sec. 3.1.2). Code section addresses which appear as immediate (imm32) operands to `mov r32, imm32` or `push imm32`, could be identified as function entry points even if they are targets of an indirect jump (non-PIC jump table targets are, however, treated specially and identified).

Function boundary identification helps prevent scenarios where the prologue is instrumented, but the epilogue is not and vice versa. Since the latter case could cause false alarms (since the epilogue checking code would be trying to find a match for the return address on the stack, but since it was never saved (no prologue instrumentation code), it won't find it in the Return Address Repository (RAR), thus flagging a false exception). We want to avoid that altogether, which can be achieved by "optimistic" identification of functions. This false identification, however could result in a function having an instrumented prologue, but an uninstrumented epilogue. Such a function, if called too frequently in a manner that it exits from an uninstrumented epilogue, then the RAR will eventually overflow, since there is no code to pop the return address off the RAR. Another potential problem due to false identification is missed attacks. If false identification causes an "entry point" to be inserted within a function body then the single function gets divided into two. Here the "second" function won't have an interesting prologue, hence all subsequent returns in this function will be missed. However, false identification of functions never jeopardizes program correctness unless, of course, an entire chunk of data misidentified as code forms a function, with both interesting prologue and epilogue, which is an extremely unlikely scenario.

In summary, PIC, indirect branches and callback functions could cause some security loopholes in the input programs to be un-protected. Empirical results show that indirect branches typically are 5-8 % of all branch instructions (Section 5.2.1, Table 5). Only a fraction of this (if at all any) could possibly result in a missed attack.

As for false alarms, they could arise due to hand crafted assembly code, mostly with inter-procedural

jumps and/or entry and exit points in different functions. An example of such a case was observed when we instrumented Microsoft Access. Here, the control jumps from Fn1 to label, which is in Fn2 and exits from Fn2.

```
Fn1: // no 'interesting' prologue
    :
    jne label
    :
    ret // no 'interesting' epilogue

Fn2: // 'interesting prologue'
    :
label:
    :
    ret // 'interesting' epilogue
```

Fn1 has an uninstrumented prologue, so its return address is not saved in the RAR and Fn2's epilogue is instrumented, so a return address check is done on exit from Fn2. The RAD epilogue of Fn2 will flag an exception, since it cannot find the on-stack return address in the RAR, thus a false alarm.

Other recipes for false alarms include data misidentified as code which looks exactly like an interesting prologue, or an entire chunk of data which appears like an interesting function, both of which are rather uncommon.

4.3.2 Potential Attacks Due to Limitations of RAD

As in RAD [1], the current binary-rewriting RAD prototype can protect applications from any kind of buffer overflow attack that corrupts the return address on the stack. Thus it can resist conventional stack smashing attacks and frame pointer based attacks [19]. However, it cannot prevent memory pointer corruption attacks [18], which do not affect the return address in any way. They simply modify the contents of the import table (Global Offset Table - GOT or Import Address Table - IAT), which makes it impossible for RAD to detect them. Fortunately, no actual network security breach incidents that are based on this type of attacks have been reported.

4.3.3 Multi-Threaded Applications

The current implementation doesn't handle multi-threaded applications. An idea to implement the solution for multi-threaded applications, comes from [26]. We can access the Thread Information Block (TIB) structure using the FS segment register. Code generated by compilers to set up exception handlers and to allocate storage for thread local variables, typically reveal this use of the FS register. The TIB contains an array of slots for thread local storage. What we could do is have a separate RAR space for each thread (taking care that RAR spaces of

Step	Size
Return Address Repository	16 Kbytes
Exception Handler	130 bytes
Installing Exception Handler	19 bytes
Set up RAD mine-zones	55 bytes
Search for VirtualProtect()	371 bytes
Total	16.2 - 16.6 KBytes

Table 1: The constant space overhead of binary-rewriting RAD. The last row corresponds to the step that searches the kernel32.dll export table for the entry point of VirtualProtect().

two threads don't bump in to each other), and store the address of the RAR in one of the thread local storage slots, which can be used by the RAD prologue and epilogue code, to figure out which RAR to work with. However, the use of the FS register, although a well-known fact in the Windows world, still falls into the category of undocumented information. There would probably be Win32 API functions, which do something like this, however the cost of invoking an API call at every RAD prologue and epilogue would be prohibitive.

4.3.4 Self-Modifying Code

Self-modifying code, like those missed functions due to indirect branches, makes control flow analysis difficult. Moreover, if a piece of code is added only at run-time to the heap, there is no way RAD can add checks to it.

5 Experimental Results

To validate the correctness of the binary-rewriting RAD prototype, we need to verify that the RAD code is injected into appropriate places in the input binary AND the RAD code does protect the input binary from buffer overflow attacks in a way that does not incur significant space overhead or run-time performance cost. In the following subsections, we present results that show that the current binary-rewriting RAD prototype does do a reasonable job in disassembly accuracy and low-overhead protection against buffer overflow attacks.

5.1 Micro-Benchmark Results

To establish the baseline performance for the binary-rewriting RAD prototype, we apply it to a set of synthetic programs and measure its space and performance overhead. Table 1 shows the constant space overhead associated with binary-rewriting RAD, which excludes the per-function prologue and epilogue RAD code. Every instrumented function needs a prologue and epilogue checking code, which take 38 and 41 bytes, respectively.

We then measure the performance overhead of an instrumented function due to its prologue and epilogue RAD code. Depending on whether an epilogue RAD code is triggered by a jump instruction or by a software

exception handler, the measured performance overhead is different. We tested three different instrumented functions:

- void fn() that does nothing and invokes prologue and epilogue RAD code through a jump instruction
- void fn() that does nothing and invokes prologue RAD code through a jump and epilogue RAD code through a software exception
- void fn() that does some amount of computation (incrementing a variable 25000 times), without making any other function calls

The performance penalty of the binary-rewriting RAD prototype is defined as: $\frac{\text{AdditionalRADOverhead}}{\text{OriginalRunTime}}$, and was measured using the Pentium performance counter, which has a resolution of 2 nsec.

For the do-nothing test function case, the overhead of RAD is 34.25%, which is higher than expected, considering that both the prolog and epilogue RAD code size is just about 9 to 11 instructions, each of which is such a simple instruction as 'push reg32', 'pop reg32', 'mov reg32, mem32', 'cmp reg32, mem32', 'add mem32, imm32' 'sub reg32, imm32' etc., none of which appear to be costly. We believe this performance overhead is due to additional instruction cache misses that arise because the code region of the test function is separate from that of its prolog and epilogue RAD code. If a function's epilogue does not contain enough space to hold a jump instruction, the epilogue RAD code is implemented inside an exception handler. The additional performance overhead due to exception delivery and return, as compared to two jump instructions, is almost quadrupled, as shown in the second test function case of Table 2. When the test function is doing something more computation-intensive, as in the third test function case, the relative performance overhead of RAD immediately becomes negligible. Compared with the original RAD system [1], which works on source code only, binary-rewriting RAD performs better in all three cases, because its prolog and epilogue code is implemented in assembly and is thus more efficient. This result is somewhat surprising as the original RAD system places per-function prolog and epilogue code together with the associated function, rather than in a separate code region, and therefore does not incur additional instruction cache miss penalty.

5.2 Macro-Benchmark Results

We experimented with a wide variety of commercial grade Windows applications, including BIND DNS server, DHCP server, a third-party FTP server, Microsoft Telnet Server, MS FrontPage, MS Publisher, MS PowerPoint, MS Access, Outlook Express, CL compiler,

Test Function	Cycle Counts - Original	Cycle Counts - RAD	Relative Penalty
Null function	292	392	34.25%
Null function + epilogue	271	641	136.53%
Incrementing function	350,425	350,722	0.085%

Table 2: *Per-function performance overhead due to the RAD code injected into an instrumented function. The Null function + epilogue case is the same as the Null function case except its epilogue is invoked through a software interrupt. The Incrementing function case corresponds to a function that increments a variable 25000 times. All measurements are in terms of Pentium cycle counts.*

MSDEV.EXE (Visual C++ development environment), Windows Help (Winhlp), and Notepad. After rewriting, all the above programs behave exactly the same as before, except MS Access, which generated a false alarm due to hand crafted assembly code (described in Section 4.3), and the third-party FTP server, which has an internal exception handler that conflicts with the debugger exception handler that binary-rewriting RAD installs. The initial experiences collected from running the binary-rewriting RAD prototype against a wide array of regular desktop applications and Internet servers, which are the prime targets for buffer overflow attacks, convinced us that this prototype is sufficiently mature to preserve the program semantics of complex production-grade applications while providing them with protection against buffer overflow attacks. Of course, more exhaustive tests are required to be absolutely sure about the accuracy of disassembly and the protection strength of RAD.

5.2.1 Disassembly Accuracy

The binary-rewriting RAD prototype uses control flow analysis and a set of other heuristics to distinguish between code from embedded data. In general, control flow analysis is quite effective in identifying the code regions for non-interactive applications, which usually do not have many call-back functions. However, for interactive GUI applications, such as those in Microsoft Office suite, control-flow analysis alone is not quite as effective because of the hidden call-back functions. Therefore these applications represent the most challenging test programs for a disassembler. Table 3 shows the disassembly accuracy of three such programs, MS Powerpoint, MS Frontpage, and MS Publisher. The disassembly accuracy of all these programs is above 99%. The way we measure disassembly accuracy is to manually inspect the resulting assembly code and determine whether the instructions look “reasonable.” From our experiences, instructions that are disassembled from data tend to appear out of place and thus can be easily detected.

Since the manual inspection method used above may not seem rigorous enough, to further verify our disassembly results, we experimented with certain Cygwin ported Unix applications (with available sources)

on Windows, compiled with gcc’s profiling options and then analyzed them offline with gprof.

Thus the % of both missed functions and unprotected returns in interesting functions appear to be reasonable. The missed functions in Apache were typically functions without any absolute address references in the code section, which were invoked through a table of function pointers to which the addresses of those functions were assigned statically. The table, being a static array variable, is located in the .data section and so were the function addresses. The static call graph generated by gprof also shows the parents of these functions as ‘unidentified’.

Because the results obtained from control flow analysis is guaranteed to be correct, it is useful to measure the percentage of instructions that can be identified through control flow analysis, which gives an indication of how useful other heuristics are in identifying instructions, especially for GUI-intensive interactive applications. Table 5 shows the total number of instruction bytes in each test application and the percentage of them that control flow analysis can successfully detect. As expected, for non-interactive applications, which rarely use any call-back functions, control flow analysis can achieve a very high detection accuracy, more than 97%. However, for interactive applications, the percentage is around 80%. The difference between the coverage percentages in Table 3 and 4 for the three programs, MS Powerpoint, MS Frontpage, and MS Publisher, represent the contribution of the pattern-based heuristics that binary-rewriting RAD employs to the total code region coverage.

Finally, because control flow analysis plays such an important role in the disassembly process, it is instructive to investigate deeper why it cannot detect all the instructions in the program. Other than functions that do not have explicit call sites, indirect branch instructions are the main culprit. We measure the percentage of indirect control branch instructions in the test applications and the results are shown in Table 5. Again, interactive applications such as MS Powerpoint and Access tend to have a higher percentage of indirect branches than others, which reflects the event-driven programming style of these applications, and correspondingly a more extensive use of function pointers and switch statements.

Application	Code section size	No. of incorrectly decoded bytes (approximation)	Accuracy
MS PowerPoint	4.059 MB	2500	99.93%
MS Publisher	2.314 MB	50	99.99%
MS FrontPage	983 KB	900	99.91%

Table 3: Disassembly accuracy achieved as measured through manual inspection of the resulting assembly code. Higher accuracy means that more bytes are successfully disassembled.

Application	No. of functions (source code)	No. of functions (disassembly)	Miss %	No. of returns unprotected	% of returns unprotected
Gzip	234	234 (80)	0	2	0.85
Wget	626	626 (140)	0	3	0.48
Apache	1191	1159 (350)	2.69	38	3.19
Whois	148	148(15)	0	0	0
OpenSSL	2820	2812(780)	0.283	7	0.248

Table 4: Evaluation of disassembly results by comparison with original program sources. The numbers in the parenthesis on the third column represent the number of falsely identified functions.

In summary, our disassembly results appear to be better than the previous best reported in the literature [22], which claims 99.9% precision using binaries with relocation information, but most of their experiments were on smaller programs, all of which were plain command line programs without any GUI callback functions, which makes disassembly tougher. Furthermore, the presence of symbol table information in binaries (possibly inadvertently) eliminates problems regarding function boundary identification. However, there is a question of the symbol table format. It could either be the generic COFF symbol table, supported by the PE/COFF binary formats, or it could be a compiler specific format, like the VC++ .pdb. Apparently, compilers tend to favor their proprietary formats for symbol table over the generic COFF format. This is evident since the default compilation options for generating debug information do not produce COFF symbol tables, and generate proprietary symbol tables instead.

5.2.2 Run-Time Overhead

An important consideration in the design of RAD is the minimization of performance overhead due to per-function prologue and epilogue RAD code. The relative performance overhead of RAD with respect to a test application is defined as

$$\frac{\text{Execution time with RAD} - \text{Execution Time w/o RAD}}{\text{Program Execution Time without RAD}}$$

The results are in Table 6, which shows the run-time performance overhead of binary-rewriting RAD for typical Internet applications is quite small, around 1%. The space overhead of binary-rewriting RAD for real appli-

cations is also quite reasonable, as shown in Table 7. The highest percentage is still smaller than 35%. Both results demonstrate that the overhead of binary-rewriting RAD is quite reasonable for practical applications, given the additional protection it provides.

Because the cost of invoking an epilogue RAD code through an exception handler is around four times as expensive as that through a jump instruction, it is important to find out how frequent epilogue RAD code is invoked through an exception handling mechanism. If it occurs frequently, then perhaps a more sophisticated mechanism needs to be developed. Table 7 also shows the percentage of functions in the test applications whose epilogue RAD code is triggered via an exception handler. The statistics in Table 7 show that the percentage of functions that do not have enough instruction space for a jump instruction is fairly low, less than 3%, which justifies our design decision of using this expensive solution in these infrequent cases. Please note that, these are results of static analysis. It is possible that, at run-time one of these functions get invoked 50% of the times, in which case the performance might get seriously hit. While it is possible to instrument binaries to report the % of functions called at run-time which need the use of the INT 3 software interrupt, we are not clear if that would say much, since at the end of the day, we can still just say that, since the % of such functions (from our experiments) is typically 0.03% to 2.5%, probabilistically the % of such functions among those called at run-time would be of a similar order.

5.3 Resilience to Buffer Overflow Attacks

The Windows help program (Winhlp32.exe) on Windows NT 4.0 with Service Pack 4 has a buffer over-

Application	% Covered by Control Flow Analysis	% of Indirect Branch Instructions
WFTpd (Ftp server)	97.13%	5.81%
BIND (DNS server)	97.32%	5.42%
MS Access	84.57%	8.29%
Notepad	97.54%	1.73%
MS Powerpoint	80.11%	7.54%
Windows Help	99.67%	1.41%
MS FrontPage	87.20%	8.98%
MS Publisher	93.86%	8.94%

Table 5: Column 2 shows the percentage of a program's code section bytes that is detected purely through control flow analysis. Column 3 shows the percentage of indirect branch instructions among all the branch instructions. RET instructions are not included in this count.

Application	Original execution time (msec)	Binary RAD execution time (msec)	% Overhead
BIND	122.56	123.85	1.05%
DHCP server	122	123.5	1.23%
PowerPoint	145	150	3.44%
Outlook Express	138.2	140	1.29%

Table 6: Whole program performance overhead due to the insertion of binary-rewriting RAD code. For BIND, the response time measurement is averaged over 10 queries issued using the client program `dig.exe`. For the DHCP server, the measurement is the startup and initialization time averaged over 6 runs. For PowerPoint, the measurement is the time taken to render a 90Kbyte presentation averaged over 6 runs. For Outlook Express, the measurement is the startup and initialization time averaged over 6 runs.

flow vulnerability, which occurs when it reads a content file (.CNT) with a very long heading string. We instrumented Winhlp32.exe using our binary-rewriting RAD tool, and the augmented binary successfully resists the attack mounted by a published exploit code [3].

6 Conclusions and Future Work

We have presented a buffer overflow defense mechanism using static binary translation based on the RAD [1] model. To the best of our knowledge, this is the first work reported in the open literature that applies static binary translation technology to a concrete application security problem. While a robust binary rewriting infrastructure, such as tools like Etch [7], does exist, published papers on these systems have never documented in detail, the design and implementation issues involved, the solutions adopted to address them and their effectiveness in a quantitative manner. Our contribution lies, not in inventing new approaches to static binary translation but in being the first study to implement state-of-the-art techniques into a working system and evaluate their effectiveness on commercial-grade Windows applications. We believe that this paper exhaustively covers most binary translation issues in substantial depth and detail and presents a comprehensive set of experimental results to demonstrate the efficacy of the design decisions we have made. Finally, the resulting binary-rewriting RAD system achieves qualified success as an important tool

to protect legacy applications whose source code is not available against buffer overflow attacks, and thus significantly broadens the applicability of buffer overflow defense mechanisms developed in the research literature. Although, it may not achieve the stated goal of providing the same level of protection as its compiler-based counterpart, in a few cases, it is primarily due to a fundamental deficiency, one that none of the known works in the binary translation literature have done better with, as far as we can tell.

Currently, we are exploring more robust and foolproof fall-back mechanisms to deal with scenarios of incorrect disassembly and lack of sufficient space for 'in place' translation. As an immediate next step, we intend to experiment our binary translation engine with Dynamically Linked Libraries (DLLs), since a major chunk of Windows services are implemented as DLLs. Finally, we aim to apply the lessons from exploring static binary translation techniques to build copy- and tamper-resistant software.

7 Acknowledgments

To Sang Cho for his open source disassembler [13] and to our shepherd, Dawn Song and the anonymous reviewers for their valuable feedback.

Application	% Increase in Size of Executable File	% of Functions Using the INT 3 Handler
WfTpd (Ftp server)	34.06%	2.57%
BIND (DNS server)	32.65%	0.00%
MS Access	11.29%	2.61%
MS Powerpoint	9.74%	0.83%
Windows Help	32.79%	0.098%
MS FrontPage	16.45%	0.031%
MS Publisher	10.84%	1.58%

Table 7: Column 2 shows the space overhead of binary-rewriting RAD for different test applications in terms of percentage increase in size of the executable file after rewriting. Column 3 shows the percentage of functions among those identified that need to invoke RAD epilogue code through the INT 3 handler

References

- [1] Tzi-cker Chiueh and Fu-hau Hsu, *RAD: A compile time solution for buffer overflow attacks*, 21st IEEE International Conference on Distributed Computing Systems (ICDCS), Phoenix, AZ, April 2001
- [2] Aleph One, *Smashing the stack for fun and profit*, Phrack Magazine 7 (49), November 1996
- [3] David Litchfield, *Windows NT buffer overruns Winhlp32*: <http://community.core-sdi.com/juliano/mnemonix-whlpbo.htm>
- [4] dark spyrit, *Win32 Buffer Overflows - Location, Exploitation and Defense*, Phrack Magazine 55 (15), May 2000
- [5] A. Srivastava and A. Eustace, *ATOM: A System for Building Customized Program Analysis Tools*, SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 196–205, June 1994.
- [6] James Larus and Eric Schnarr, *EEL: Machine-independent executable editing*, SIGPLAN Conference on Programming Languages, Design and Implementation, pages 291–300, June 1995.
- [7] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad. *Instrumentation and optimization of win32/intel executables using Etch*. In USENIX Windows NT Workshop, 1997.
- [8] *LEEL*, <http://www.geocities.com/fasterlu/leel.htm>
- [9] C. Cifuentes and M. Van Emmerik, *UQBT: Adaptable Binary Translation at Low Cost*, IEEE Computer, March 2000.
- [10] Crispin Cowan et al., *Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks*, 7th USENIX Security Symposium, San Antonio, TX, January 1998.
- [11] Microsoft compiler extension for buffer overflow defense, <http://go.microsoft.com/fwlink/?Linkid=7260>
- [12] Stackshield, www.angelfire.com/sk/stackshield/
- [13] Win32 Disassembler, www.geocities.com/sangcho
- [14] Hiroaki Etoh. *GCC extension for protecting applications from stack-smashing attacks*. <http://www.trl.ibm.co.jp/projects/security/ssp>
- [15] CASH: *Checking Array Bound Violation Using Segmentation Hardware*, <http://www.ecsl.cs.sunysb.edu/softsecure/project.html>
- [16] R. Jones and P. Kelly, *Backwards-compatible bounds checking for arrays and pointers in C programs*, <http://www-ala.doc.ic.ac.uk/phjk/BoundsChecking.html>
- [17] Intel Architecture Software Developer's Manual: Volume 3: System Programmer's Guide
- [18] Bulba and Kil3r. *Bypassing StackGuard and StackShield*. Phrack, 5(56), May 2000.
- [19] Phrack Magazine 55 (8), May 2000: Klog - The frame pointer overwrite
- [20] Arash Baratloo, Timothy Tsai, and Navjot Singh, *Transparent run-time defense against stack smashing attacks*, USENIX Annual Technical Conference, June 2000.
- [21] Vladimir Kiriansky, Derek Bruening, Saman Amarasinghe, *Secure Execution Via Program Shepherding*, 11th USENIX Security Symposium, August 2002, San Francisco, California.
- [22] Benjamin Schwarz, Saumya Debray, Gregory Andrews, *Disassembly of executable code revisited*, Working Conference on Reverse Engineering, Oct 2002.
- [23] C. Cifuentes, M. Van Emmerik, *Recovery of Jump Table Case Statements from Binary Code*, International Workshop on Program Comprehension, May 1999
- [24] Galen Hunt and Doug Brubacher, *Detours: Binary Interception of Win32 Functions*, 3rd Usenix NT Symposium, Seattle, July 1999.
- [25] Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, MSDN magazine, Feb 2002
- [26] Matt Pietrek, *Under the Hood*, Microsoft Systems Journal, 11(5), May 1996.
- [27] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam and Tia Newhall, *The Paradyn Parallel Performance Measurement Tools*, IEEE Computer 28, 11, pp.37-46 (November 1995).
- [28] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel and Ravishankar K. Iyer, *Compiler and Architecture Support for Defense against Buffer Overflow Attacks*, 2nd Workshop on Evaluating and Architecting System Dependability (EASY), San Jose, CA, October, 2002

Kernel Support for Faster Web Proxies

Marcel-Cătălin Roșu

Daniela Roșu

IBM T.J. Watson Research Center

P.O. Box 704, Yorktown Heights NY 10598, USA

{rosu,drosu}@watson.ibm.com

Abstract

This paper proposes two mechanisms for reducing the communication-related overheads of Web applications. One mechanism is *user-level connection tracking*, which allows an application to coordinate its non-blocking I/O operations with significantly fewer system calls than previously possible. The other mechanism is *data-stream splicing*, which allows a Web proxy application to forward data between server and client streams in the kernel with no restrictions on connection persistency, object cacheability, and request pipelining. These mechanisms remove elements that scale poorly with CPU speed, such as context switches and data copies, from the code path of Web-request handling.

The two mechanisms are implemented as Linux loadable kernel modules. User-level connection tracking is used to implement *uselect*, a user-level select API. The Squid Web proxy and the Polygraph benchmarking tool are used in the evaluation. With Polymix-4, a realistic forward proxy workload biased towards cache hits and small files, the reductions in CPU overheads relative to the original Squid (with *select*) are 52-72% for *uselect*, up to 12% for *splice*, and 58-78% for the two mechanisms combined. Relative to Squid with */dev/epoll*, *uselect* provides 50% overhead reduction.

1 Introduction

The advent of the World Wide Web has motivated a large body of research on improving Web server performance. Work has focused on improving the performance of the TCP/IP stack [25] (e.g., NewReno, SACK, Limited Transmit), of the Web server architecture (e.g. ZEUS[43], Apache [2], Flash[30], Squid[27], SEDA[41]), and of the interface between them (e.g., *select*[4], */dev/epoll*[34], *sendfile*[24]).

In spite of the recent progress, the ability of existing operating system architectures to handle communication intensive workloads remains limited for server 'in-the-middle' configurations, such as Web proxies, CDN servers, and Edge Servers. First, these servers handle a

large number of concurrent connections: to reduce transfer latencies for both cache hits and misses, proxies have to keep open connections to as many clients, peer caches, and origin servers as possible [8]. Second, a significant ratio of the requests arriving at forward Web proxies require transfers from origin servers or peer caches [42]. In serving these requests, content is transferred from server to client connections by copying it twice between kernel and application address spaces. Reverse Web proxies perform a similar processing, but cache misses represent a smaller fraction of their load.

Commercial operating systems include limited support for high-performance user-level Web proxies. Besides *sendfile*, the event notification mechanism */dev/epoll* is being considered for inclusion in the Linux 2.6 kernel and a *splice* service for TCP connection tunneling is included in the AIX 5.1 kernel. However, the existing support is not sufficient, and, as a result, network appliances are used in many high-traffic proxy installations. Appliances are carefully optimized for I/O intensive workloads, but compared to general-purpose servers, have higher costs and limited flexibility. We submit that extending general-purpose operating systems with support targeted for Web proxy caches will boost the performance of off-the-shelf Web proxy applications and cache infrastructures, like Squid [27] and IR-Cache [28], respectively.

This paper proposes enhancing general-purpose operating systems with two mechanisms, *user-level connection tracking* and *data stream splicing*. These mechanisms enable Web applications to reduce their communication-related overheads by reducing the number of system calls and the amount of data copied between user and kernel domains, operations that are known to scale poorly with processor speeds [1, 29].

User-level connection tracking allows an application to coordinate its non-blocking network operations and to monitor the state of its connections with minimal switching between application and kernel domains. The mechanism is based on a shared memory region between kernel and application in which the kernel propagates elements of the application's transport and socket-layer

state, such as the existence of data in receive buffers or of free space in send buffers. The application can access these state elements directly, without system calls. The mechanism is secure as only information pertaining to the application's connections is provided in its memory region. The mechanism can be used to implement low-overhead versions of the `select/poll` APIs, as well as new connection-tracking APIs. For instance, with socket-buffer availability propagated as actual number of bytes, applications can perform more efficient I/O by issuing I/O operations only when the number of bytes that can be transferred is greater than a specified threshold. Similarly, with transport-layer state like congestion window size and round-trip time, applications can learn about the latency characteristics of their connections and selectively customize their replies to improve the client response times [18].

Data-stream splicing allows an application to perform data forwarding between corresponding server and client TCP streams in the kernel, at the socket level. This proposal is the first to address the whole range of data transfer operations performed by a Web proxy cache application, supporting persistent connections, cacheable content, pipelined requests, and tunneled transfers. This mechanism draws significant benefits from the decision to implement it at socket level. Compared to user-level data forwarding, the mechanism eliminates data copies and system calls [11, 37]. Compared to IP-level alternatives [21, 39], the mechanism can be applied to data streams with different TCP connection characteristics (e.g., SACK) and it provides the application with full and efficient control over unsplicing and payload caching.

The two mechanisms are implemented in Linux and are evaluated using Squid [27], a popular Web proxy cache application, and Polygraph [40], a benchmarking tool for Web proxies. User-level connection tracking is used to implement a user-level wrapper for the native `select` system call, called `uselect`. Microbenchmarks demonstrate that `uselect` enables reductions in CPU utilization of 60-95% relative to `select` and of 20-90% relative to `/dev/epoll` for 4-128KByte objects and 100% cache hits. Data-stream splicing enables overhead reductions of 10-70% for a workload with 100% cache misses. With Polymix-4, a realistic forward-proxy workload biased towards small file sizes and cache hits, the reductions in CPU overheads relative to the original Squid (using `select`) are 52-72% for user-level `select`, up to 12% for splice, and 58-78% for the two mechanisms combined.

While our mechanisms have been proposed and evaluated in the context of Web proxies, they can benefit a wider range of applications. CDNs, Edge Servers, and internet applications based on SEDA [41] can benefit from low-overhead access to the socket- and network-

layer state of their connections. Peer-to-peer infrastructures that include content forwarding, like Squirrel [14], can use the data-stream splice to lower node overheads.

Modular implementations of the networking stack and the socket layer enable simple implementations for the two mechanisms as loadable extension modules. Our approach is to replace methods of the transport and socket layers with new implementations, or with wrappers for the original methods. The Linux implementation presented in this paper does not require modifications of the kernel source tree.

The remainder of this paper is organized as follows. Sections 2-3 describe the proposed mechanisms. Section 4 describes the experimental testbed and methodology. Section 5 presents the results of the experimental evaluation. Section 6 discusses related research and Section 7 summarizes our contributions.

2 User-level Connection Tracking

Experiences with communication-intensive applications, such as Web servers, demonstrate that restricting the number of kernel threads used by the application is critical to achieving good performance. The most efficient architectures are event-driven [27, 30] because, by avoiding blocking I/O operations, they handle a large number of connections with a small number of control threads. Efficient non-blocking I/O requires a mechanism for tracking connection state, such that I/O operations are issued only when guaranteed to be successful.

The traditional OS mechanisms for connection-state tracking, `select` and `poll`, retrieve connection state from the kernel by performing two context switches and two data copy operations; the amount of data copied is proportional to the number of existing connections. Recently proposed event delivery mechanisms [5, 19, 34] allow more efficient in-kernel implementations, avoid the application-to-kernel data copy, and even the kernel-to-application copy [33, 34]. However, the benefits of these optimizations are partially offset by the possible increase in the number of system calls since the application has to register and cancel its 'interests' for every socket. The additional system calls are shown to represent a relatively high overhead for sockets with short lifetimes [19].

The *user-level connection tracking* mechanism proposed in this paper attempts to further reduce the number of system calls related to connection-state tracking and to extend the set of connection-state elements that applications can exploit. The approach is to propagate certain elements of a connection's socket- and/or transport-layer state at the user level, in a memory region shared between the kernel and the application (see Figure 1). The application can retrieve the propagated state using memory read operations, without any context switches and data

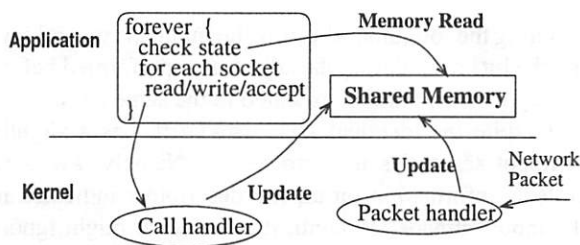


Figure 1: User-level connection tracking.

copies. The mechanism is secure because each application has a separate memory region which contains only information pertaining to the application's connections. The mechanism does not require any system calls for connection registration/deregistration. All connections created after the application registers with the mechanism are automatically tracked at user-level until closed.

The mechanism allows for multiple implementations, depending on the set of state elements propagated at user level. For instance, in order to implement the *select/poll*-type connection state tracking APIs, the set includes elements that describe the states of send and receive socket buffers. Similarly, the representation of state elements in the shared memory region depends on the implementation. For instance, for the *select/poll*-type tracking, the representation can be a bit vector, with bits set if read/write can be performed on the corresponding sockets without blocking, or it can be an integer vector, with values indicating the number of bytes available for read/write.

The same set of state elements is associated with all of the application's connections. The data structures in the shared region should be large enough to accommodate the maximum number of files a process can open. However, the shared memory region is typically small. For instance, an application with 65K concurrent connections and using 16 bytes per connection requires a 1MByte region, which is a small fraction of the physical memory of an Internet server.

In addition to direct memory reads, applications can access the shared memory region through user-level library calls. For instance, when the shared state includes information on socket-buffer availability, the application can use user-level wrappers for *select/poll*. Such wrappers can return a non-zero reply using only the information in the shared region; otherwise, if parameters include file descriptors not tracked at user level or a non-zero timeout, the wrappers fall back on the corresponding system calls.

The kernel component updates the shared memory region during transport and socket layer processing, and at the end of read and write system calls (see Figure 1). The shared region is not pageable and updates are implemented using atomic memory operations. The cost of

updating variables in the shared memory region is a negligible fraction of the CPU overhead of sending/receiving a packet or of executing a read/write system call.

The kernel component exploits the modular implementation of the socket and transport layers. In Linux, the socket layer interface is structured as a collection of function pointers, aggregated as fields of a 'struct *proto_ops*' structure. For IPv4 stream sockets, the corresponding variable is '*inet_stream_ops*'. This is accessible through pointers from each TCP socket and includes pointers to the functions that support the read, write, *select/poll*, accept, connect, and close system calls. Similarly, the transport layer interface is described by a struct *proto* variable called '*tcp_prot*', which includes pointers for the functions invoked upon TCP socket creation and destruction. Also, each TCP socket is associated with several callbacks that are invoked when events occur on the associated connection, such as packet arrival or state change.

In order to track a TCP connection at user level, the kernel component replaces some of these functions and callbacks; the replacements capture socket state changes, filter and propagate them in the shared region. Connection tracking starts upon return from the *connect* or *accept* system calls. To avoid changing the kernel source tree, in this implementation, the tracking of accept-ed connections starts upon return from the first *select/poll* system call.

User-level Tracking with *select* API. In this paper, we use the proposed connection-state tracking mechanism to implement *uselect*, a user-level tracking mechanism with the same API as *select*.

For this implementation, the shared memory region between kernel and application includes four bitmaps: the *Active*, *Read*, *Write*, and *Except* bitmaps. The *Active* bitmap, A-bits, records whether a socket/file descriptor is tracked, i.e., monitored, at user level. The *Read* and *Write* bitmaps, R- and W-bits, signal the existence of data in receive buffers and of free space in send buffers, respectively. The *Except* bitmap, E-bits, signals exceptional conditions.

The implementation comprises an application-level library and a kernel component. The library includes (1) *uselect*, a wrapper for the *select* system call, (2) *uselect_init*, a function that initializes the application and kernel components and the shared memory region, and (3) *get_socket_state*, a function that returns the read/write state of a socket by accessing the corresponding R- and W-bits in the shared region.

The *uselect* wrapper, consisting of about 650 lines of C code, is composed of several steps (see Figure 2). First, the procedure checks the relevant information available at user level by performing bitwise AND between the bitmaps provided as parameters and the

```

int uselect(maxfd, readfds, writefds,
            exceptfds, timeout) {
    static int numPass = 0;
    int nbits;
    nbits = BITS_ON(readfds& R-bits& A-bits)
        + BITS_ON(writefds& W-bits& A-bits)
        + BITS_ON(exceptfds& E-bits& A-bits);
    if(nbits > 0 && numPass < MaxPass) {
        adjust readfds, writefds, exceptfds
        numPass++;
    } else {
        adjust & save maxfd, readfds, writefds,
            exceptfds
        nbits = select(maxfd, readfds, ...)
        numPass = 0;
        if( proxy_socket set in readfds) {
            check R/W/E-bits
            adjust nbits, readfds, writefds,
                exceptfds
        }
    }
    return nbits;
}

```

Figure 2: User-level select.

shared-memory bitmaps. For instance, the `readfds` bitmap is checked against the A- and R-bitmaps. If the result of any of the three bitwise ANDs is nonzero, `uselect` modifies the input bitmaps appropriately and returns the total number of bits set in the three arrays; otherwise, `uselect` calls `select`. In addition, `select` is called after a predefined number of successful user-level executions in order to avoid starving I/O operations on descriptors that do not correspond to connections tracked at user level (e.g., files, UDP sockets).

When calling `select`, the wrapper uses a dedicated TCP socket, called *proxy socket*, to communicate with the kernel component; the proxy socket is created at initialization time and it is unconnected. Before the system call, the bits corresponding to the active sockets are masked off in the input bitmaps, and the bit for the proxy socket is set in the read bitmap. `maxfd` is adjusted accordingly, typically resulting in a much lower value; `timeout` is left unchanged. When an I/O event occurs on any of the 'active' sockets, the kernel component wakes-up the application which is waiting on the proxy socket. Note that the application never waits on active sockets, as these bits are masked off before calling `select`. Upon return from the system call, if the bit for the proxy socket is set, a search is performed on the R-, W-, and E-bit arrays. Using a saved copy of the input bitmaps, bits are set for the sockets tracked at user level and whose new states match the application's interests.

The `uselect` implementation includes optimizations not shown in Figure 2 for simplicity. For instance,

counting the 'on' bits, adjusting the input arrays, and saving the bits reset during the adjustment performed before calling `select` are all executed in the same pass.

Despite the identical API, `uselect` has a slightly different semantics than `select`. Namely, `select` collects information on all file descriptors indicated in the input bitmaps. In contrast, `uselect` might ignore the descriptors not tracked at user level for several invocations. This difference is rarely an issue for Web applications, which call `uselect` in an infinite loop.

The `uselect` kernel component is structured as a device driver module, consisting of about 1500 lines of C code. Upon initialization, this module modifies the system's `tcp_prot` data structure, replacing the handler used by the socket system call with a wrapper. For processes registered with the module, the wrapper assigns to the new socket a copy of `inet_stream_ops` with new handlers for `recvmsg`, `sendmsg`, `accept`, `connect`, `poll`, and `release`.

The new handlers are wrappers for the original routines. Upon return, these wrappers update the bitmaps in the shared region according to the new state of the socket; the file descriptor index of the socket is used to determine the update location in the shared region.

The `recvmsg`, `sendmsg`, and `accept` handlers update the R-, W-, or E-bits under the same conditions as the original `poll` function. In addition, `accept` assigns the modified copy of `inet_stream_ops` to the newly created socket.

Replacing the `poll` handler, which supports `select/poll` system calls, is necessary in our Linux implementation because a socket created by `accept` is assigned a file descriptor index *after* the return from the `accept` handler. For a socket of a registered process, the new `poll` handler determines its file descriptor index by searching the file descriptor array of the current process. The index is saved in an unused field of the socket data structure, from where it is retrieved by event handlers. Further, this function (1) replaces the socket's `data_ready`, `write_space`, `error_report`, and `state_change` event handlers, and (2) sets the corresponding A-bit, which initiates the user-level tracking and prevents future `poll` invocations. On return, the handler calls the original `tcp_poll`.

The `connect` handler performs the same actions as the `poll` handler. The `release` handler reverses the actions of the `connect/poll` handlers.

The event handlers update the R-, W-, and E-bits like the original `poll`, set the R-bit of the *proxy socket*, and unblock any waiting threads.

Exploiting `uselect` in Squid. In order to use `uselect`, Squid is changed as follows. During initialization, before creating the `accept` socket, Squid invokes

`uselect_init`; as a result, the accept socket is tracked at user level. In each processing cycle, Squid invokes `uselect` instead of `select` to determine the states of all of its sockets. Finally, when trying to prevent starvation of the accept socket during a processing cycle, Squid uses `get_socket_state` instead of `select` to check the ready-to-read state of this socket.

Overall, `uselect` enables Squid to eliminate a significant number of systems calls with very few code modifications. Furthermore, `uselect` reduces the overhead of the remaining `select` system calls through the use of the proxy socket.

3 Data-Stream Splicing

The data-stream splicing mechanism proposed and evaluated in this paper enables a Web proxy to forward data between its server and client connections in the kernel, with support for content caching, persistent connection, and pipelined requests. The mechanism helps reduce the number of context switches and data copy operations incurred when serving cache misses, POST requests, and connection tunnels.

In its basic functionality, the mechanism establishes, in the socket layer, a data path between two data streams, such that packets received on one stream are forwarded on the other stream immediately, in interrupt context. On servers with zero-copy networking stacks and adapter support for checksum computation, the payload of forwarded packets is not touched by the proxy CPU.

The proposed mechanism extends previous proposals [22, 37, 39] with support for the following functionality:

- request pipelining and persistent connections;
- content caching decoupled from client aborts;
- efficient splicing for short transfers.

The new socket-level splicing mechanism can establish bi- and unidirectional data paths. Figure 3 illustrates the corresponding data flows. In the bidirectional mode, the traditional model for in-kernel splicing [21, 22, 37], data received on either of the two connections is forwarded to the peer endpoint. Connection close events are also forwarded. Splicing terminates when both connections are in `CLOSE` state. Bidirectional splice is typically used for SSL tunneling; it cannot support request pipelining and persistent connections.

The unidirectional mode addresses these limitations. Data coming from one endpoint, the *source*, is forwarded on the peer connection, towards the *destination*, while data coming from *destination* is provided to the application and not forwarded to the *source*. Connection close events are not forwarded. Splicing terminates (1) after transferring a specified number of bytes, or (2) when the *source* closes the connection.

Unidirectional mode is typically used to support re-

quest pipelining and persistent connections for HTTP GET and POST. For instance, for an HTTP GET, the *source* is the origin server, and the *destination* is the client; the server response is forwarded inside the kernel from the server to the client connection, while additional client requests are handled by the application and, if necessary, forwarded to the server. After `unsplice`, the same client connection can be used to transfer cached objects or it can be spliced with a connection to a different server.

Optionally, in unidirectional mode, a copy of the transferred payload is provided to the application. This mode, called *KeepCopy*, enables a Web proxy to populate its cache while exploiting kernel-level data forwarding. The application receives the content via the traditional `read` interface. The input stream is not terminated if the client aborts its connection, thus the cache operation can be completed.

Our experience with Web proxy workloads led us to develop a splice API that minimizes system call overheads, even for short transfers. The approach is to allow the application to combine several system calls that are typically issued in sequence by the application, and to eliminate some of the remaining system calls. Specifically, the basic splice interface, which specifies the connections, the type of splicing, and the termination condition, can be combined with: (1) a write to the client connection, used for the HTTP headers and the first content segment, and (2) a read from the server connection in *KeepCopy* mode. In addition, an application can save a system call when not interested in acquiring the amount of forwarded data returned by the `unsplice` command. Namely, with the *AutoRelease* option set in the splice request, the kernel releases the splicing context upon termination, eliminating the need for an explicit `unsplice` command from the application. Also, in *KeepCopy* mode, the application can specify a minimum input size, which is used to reduce the number of read system calls.

Implementation. The implementation of data-stream splice, about 4000 lines of C code, is structured as a loadable module. The module maintains a description and state for each spliced connection. Connection descriptors are maintained in a hash table with hash entries computed from the addresses of related TCP sockets.

The application uses `ioctl` calls to issue *Splice* and *Unsplice* commands to the kernel. Parameters and results are represented in a *SpliceRequest* data structure. For *Splice*, this data structure identifies the two connections (*fd0*, *fd1*), the type of splicing (e.g., bi- or unidirectional, termination type, with or without *KeepCopy* and *AutoRelease*), and the data to send on *fd1* before splicing is initiated. For unidirectional splice, *fd0* is the source and *fd1* is the destination. The data structure also includes parameters used in one or more of the splicing modes, such as the payload limit, when termination is based on

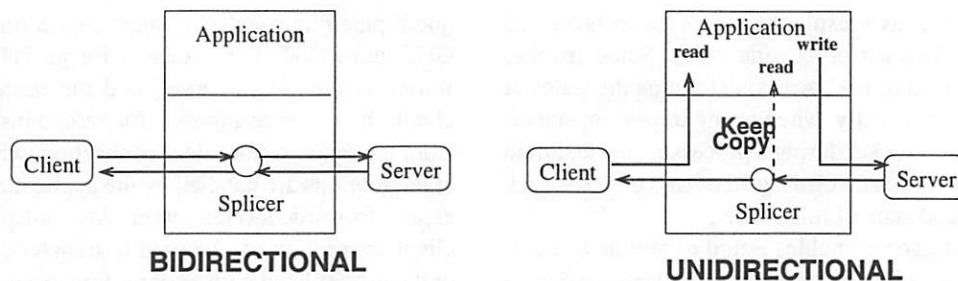


Figure 3: Types of splice interactions.

the transferred amount. For *KeepCopy*, the application can indicate the minimum input size, (*min2read*), the maximum amount of data forwarded and not yet read (*max2cache*), and an input buffer, to be filled by the *Splice* command if data is already available. *max2cache* helps prevent slow applications from overloading the kernel buffer cache.

After the *Splice* command, the application waits for notification of splicing termination, provided by the kernel as a *POLLOUT* event on the destination connection. This occurs when the transfer reaches the payload limit or the *FIN* packet is processed, and, for *KeepCopy*, when the application reads the last segment of the forwarded payload. If termination is due to connection abort, an error condition is signaled to the user.

For short transfers, the splicing can terminate in the *Splice* call. In this case, the application is informed by an appropriate return code and the splice context is released.

The application uses the *Unsplice* command to release the splicing context after termination notification or when it wants to abort the splice, such as on timeout or application shut-down. The *Unsplice* request specifies the two connections and it returns the number of bytes forwarded in each direction.

In the kernel, the splicing is established if the connections are in the *ESTABLISHED* or *CLOSE_WAIT* states. Upon splicing, the two TCP sockets are assigned new *data_ready*, *write_space*, *error_report*, and *state_change* handlers and a new *inet.stream_ops* data structure, with the *recvmsg*, *sendmsg*, and *poll* entries replaced.

The new *data_ready* and *write_space* handlers, invoked when data and ACKs are received, perform most of the data forwarding. Packet forwarding is started at end of the *Splice* call and it is driven by data and ACK packet arrivals on the two connections. Forwarding never stalls except for *KeepCopy* transfers when the amount of data forwarded and not read reaches *max2cache*. In this case, forwarding is restarted at the end of *recvmsg*.

For lower overheads, the TCP checksum is computed incrementally, from the old value, when forwarding an entire packet; it is computed from scratch only when for-

warding a fraction of the packet, as it might occur at the beginning and the end of splicing.

The *recvmsg* and *sendmsg* handlers in the new *inet.stream_ops* are replaced with error-returning handlers when the corresponding operations are not permitted for the spliced socket. For instance, in bidirectional splicing, both handlers are replaced for both sockets. For unidirectional splicing, write is not permitted for the destination socket, and read is not permitted for the source socket unless splicing with *KeepCopy*.

For *KeepCopy*, the application uses the *read* system call, supported by the *recvmsg* handler, to retrieve a copy of the forwarded data. A local copy is created by cloning the packets, i.e., the *sk_buff*'s, and adding them to a list in the splice descriptor of the source socket. The new *poll* returns a *POLLIN* event when the local copy reaches the specified minimum input size (*min2read*) or when no more data is to be forwarded. The new *recvmsg* of the source socket transfers the data from the cloned *sk_buff*'s to the application buffer.

Using the mechanism in Squid. The original Squid 2.4 implementation handles cache misses as follows. The *http* module reads from a server connection in a 85KByte buffer allocated on the call stack. The data is transferred to the *store* module, which copies it in 4KByte memory blocks and notifies the *client-side* module of the new data arrival. The *client-side* processes the reply, generating the HTTP headers. If it can fill a 4KByte block, it copies data from the *store* into a send buffer, and it registers for a ready-to-write notification on the client socket. When the notification is received, the block is written, the completion routine updates the state, and the client registers to receive the next block from the *store*.

In order to exploit the data-stream splice mechanism for GET requests, we made the following changes. The *http* reads the first segment of a server reply in a 2896-byte buffer, attempting to consume only the first 2 MTUs. This size is chosen because: (1) a multiple of MTU size minimizes checksum overheads by enabling incremental checksum computation; (2) the included content and the proxy HTTP headers are below the 4Kbyte limit that can be sent in one operation by the *client-side*; and (3) a large

fraction (approx. 40%) of the objects in proxy workloads can be received in one read operation.

After the read, if HTTP headers indicate that more data is expected, the *http* initializes a splice descriptor before sending the content to the *store* and does not prepare for a new read from the server. The splice descriptor is attached to the *StorageEntry* data structure, which is reachable from *http* and *client-side* request descriptors.

For a request with a splice descriptor, the *client-side* does not wait to fill the first 4KByte block before preparing for output. When it receives the ready-to-write notification on the client socket, it invokes the *Splice* command to write the available content and to initialize the in-kernel transfer; it does not register with the *store* to receive the rest of the content.

The *Splice* parameters define a unidirectional transfer, with the server connection as source and the client connection as destination. Splicing termination is set for a payload limit, if *ContentLength* is defined, or, otherwise, for the close of the server connection. The *AutoRelease* flag is set. If the content is cacheable, the *KeepCopy* flag is set and the related parameters are defined.

If the *Splice* command returns without error or indication of termination, the client socket is registered for (1) ready-to-write notification, to process the splicing termination, and for (2) ready-to-read notification, to receive the next request from the client. In *KeepCopy* mode, the server socket is registered for ready-to-read notification. Splice-related flags are set in both client and server socket descriptors in the global *fd_table* to ensure that spliced connections are handled appropriately in the close procedure, such as when called from timeout handlers. If *Splice* returns an error, the transfer resumes at application level.

The Squid handler invoked upon splicing termination performs the following actions. First, it checks if splicing termination was abnormal, checking the server socket EOF condition for *KeepCopy*, and the socket error otherwise. Next, it invokes the *client-side* procedure for write completion, providing the size of the spliced transfer. This restarts the activity on the client connection, activating the output for the next pending request or closing the connection. Finally, the server connection is added to the pool of persistent connections or closed, according to the request parameters. If splicing termination is abnormal, both connections are closed.

On a timeout, the handler issues an *Unsplice* command, calls the *client-side* write completion procedure, and closes both connections. We have also extended Squid to splice HTTP CONNECTs and POSTs. Details are available in [38].

Overall, with the proposed splice mechanism, a Web proxy can move the 'data path' for its cache misses (i.e., HTTP body transfers) into the kernel. This releases re-

sources for the 'control path' (i.e., HTTP header processing) and cache management, which remain at user level.

4 Experimental Environment

The experimental evaluation of the two mechanisms proposed in this paper is conducted with Squid, a popular Web proxy application, and with Polygraph [40], a benchmarking tool widely used by the industry.

4.1 Hardware, System Software, and Apps

Our testbed comprises five nodes: three Polygraph nodes (two clients and one server), the Squid proxy, and a wide-area network emulator. Except for the WAN emulator which runs PicoBSD 0.445, all nodes run the Linux 2.4 kernel. An additional node, running the monitoring and data collection applications, is attached to the testbed. Table 1 describes the hardware configurations.

Table 1: Hardware configuration of the testbed.

	CPU Type	Speed (MHz)	Memory (MBytes)
PolyClient 1	PentiumIII	550	128
PolyClient 2	PentiumIII	667	256
PolyServer	PentiumPro	2x200	224
Web Proxy	PentiumIII	1000	512
WAN Emulator	PentiumPro	200	128

Except for the Polygraph server, all nodes are attached to a Gigabit Ethernet switch, the Alteon ACEswitch 180. The Polygraph server is attached via the dual-homed WAN emulator. The proxy node uses an Alteon Gigabit Ethernet adapter; the other nodes use Fast Ethernet adapters. The network links and the WAN emulator are never overloaded during the experiments. The network configuration is similar to that used in Polygraph Cache-Offs [23], with clients and servers on different subnets. **Polygraph.** Web Polygraph 2.7.6 [40] is a benchmarking tool for caching proxies and other Web intermediaries. Polygraph includes high-performance HTTP clients and servers, realistic traffic generation and content simulation, and standard workloads. These standard workloads, including the Polymix-4 used in our evaluation, are widely accepted in the web caching community.

Each client and server node (agent) runs one or more 'robots', each robot handling one connection at a time, and possibly sending several requests in a connection. A client robot maintains a predefined request rate (e.g., 0.4 req/s). The overall request rate is determined by the number of client nodes, number of robots in a client node, and per-robot request rates.

In our experiments, we use Polygraph 2.7.6, modi-

fied to allow client and server applications to open up to 12,000 concurrent connections instead of the original 1024 limit; the kernel connection limit is set accordingly.

Proxy Application. The proxy application is Squid 2.4, extended to exploit the `splice` and `uselect` interfaces proposed in this paper, as well as the `/dev/epoll` interface used for comparison with `uselect`. Squid is a typical example of an event-driven application that can manipulate a very large number of communication streams. Squid is built around an infinite `select/poll`-loop. In each cycle, the application performs input and output operations on its active sockets, depending on connection and request processing states.

We modified Squid to support up to 64K file descriptors, more than the 1024 preset limit in Linux. In addition, we made several changes to improve its scalability under high load. These changes are orthogonal to the mechanisms evaluated in this paper and are deemed necessary because of our interest in driving the application at higher loads than previously published evaluations [40] on comparable hardware. For instance, we reduced the number of calls for memory allocation by extending the use of pre-allocated buffers.

Squid can use several models of disk cache management. In our experiments, we use the *diskd* and the *null* models. The *diskd* model uses daemons to perform the (blocking) disk I/O operations, one daemon for each disk. Squid communicates with a daemon through two message queues and a shared memory region; the message queues are used for operation descriptors and completion notifications, and the shared memory is used for the data blocks subject to I/O operations.

The *null* model emulates an infinite size, 0-overhead disk cache. There is no disk I/O and the list of cacheable objects read from the server is maintained in memory.

For `/dev/epoll`, we use *ep-patch-2.4.18-0.32* [20]. `/dev/epoll` is an efficient event-delivery mechanism which uses a shared memory between application and kernel to eliminate the data copy of notification results. However, it does not eliminate the system calls for event notification retrieval, and requires system calls for socket registration and deregistration. In order to use this interface, we define a new Squid procedure for connection-state tracking similar to the one used for `poll`, and we extend the `fd_table` to include flags for read and write availability. The new procedure performs the following steps. First, it traverses the file descriptor table, identifying the sockets in which the application is interested to read or write. For each of these sockets, the event types of interest are saved. Also, sockets not registered with `epoll` are registered at this time, indicating interest in all types of events; their new read- and write-availability flags are set. Second, `epoll` is invoked, and if there is any returned list of events, this is used to set

the read/write availability flags of the indicated sockets. Third, the list of sockets in which the application has interest is traversed, and I/O operation(s) are performed if the corresponding availability flags are set. Availability flags are cleared when the corresponding I/O operations return blocking indications (i.e., `errno` is `EAGAIN`). Sockets are unregistered with `epoll` just before they are closed. The two traversals in steps one and three are also performed when using `uselect`, `select`, and `poll`.

WAN Emulation. To simulate WAN conditions, we use the same tools and settings as the Polygraph Cache-Offs. The WAN emulation tool is DummyNet [36]. For the proxy-server link, this tool introduces round-trip packet delays of 80 ms and packet losses of 0.05%. No delays or losses are introduced on the client-proxy links.

4.2 Experimental Methodology

The goal of our experimental study is to evaluate the performance of the proposed mechanisms and compare them with related mechanisms. The evaluation focuses on performance metrics like CPU utilization and response time. Towards this end, we use (1) microbenchmarks, in which the workload includes fixed size objects and the Web proxy does not perform disk I/O operations, and (2) realistic experiments, in which the workload is Polymix-4, a Polygraph workload representative for Web proxy caches, and the proxy stores cached objects on disks. Taking a high-level view at the benefits of these mechanisms, we do not attempt to quantify the individual components of the associated overhead reduction, such as data copies and context switches.

In microbenchmarks, we vary: (1) object cacheability, (2) hit ratio, (3) object size, (4) request rate, and (5) number of concurrent connections. In an experiment, all requests are HTTP GETs for objects of identical size and cacheability type (i.e., either cacheable or non-cacheable). The set of object sizes includes 4, 8, 12, 25, 64, and 128KBytes. The selection is related to the distribution of file sizes in Polymix-4, in which, with approximation, 4KBytes is the 50-th percentile, 8KBytes is the 75-th percentile, and 25KBytes is the 95-th percentile. Hit ratios are either 0% or (almost) 100%. Squid uses the *null* disk manager. Each data point represents three or more samples. For each sample, we collect statistics for 15min, after a 10min warm-up.

In the Polymix-4 experiments, we vary only the request rate. These experiments are similar to the Fourth Polygraph Cache-Off benchmarking [23], but with shorter phases. Namely, each experiment starts with an empty cache and takes 4h 30min. The fill phase runs at 160req/s for 90min. The first peak 'daily' load phase takes 25min, and the measurement peak 'daily' load phase takes 120min; the rest of the time is spent

in ramp-up and down phases. Squid uses the *diskd* disk manager, with 4 disks, each with 3GByte of caching space. Each data point represents one sample.

In all experiments, request rates are selected such that client and server nodes are never overloaded.

For each experiment, we collect the statistics produced by the Polygraph agents, by *vmstat* and *tcpstat* running on the proxy node, and by Squid. Polygraph statistics include request rates and response times. *vmstat* provides information on CPU utilization, and *tcpstat* provides information on TCP transfers. Squid provides various statistics, such as rate of cache hits and number of open sockets.

All the plots of performance metrics for a single configuration include 90-percent confidence intervals, calculated with the T-student distribution. Note that for some experiments, the confidence intervals are very small, hardly visible on the plots. This is due to the workload model and the large number of requests in each run.

Polygraph Parameters. In all experiments, except for the parameters specified above, the Polygraph testbed is configured as for the Polymix-4 workload. Among its parameters we mention: client robot request rate of 0.4 req/s, and server delays normally distributed with a 2.5s mean and 1s deviation. The number of requests that a robot sends in a connection is drawn from a Zipf(64) distribution. Similarly, the server uses a Zipf(16) distribution to close active connections. The Polymix-4 workload has 58% hit rate, and about 70% cacheability ratio.

Squid Configuration. In all experiments, Squid runs with the default configuration, except for a few changes. No access log is maintained and no access control is performed, as in the Squid evaluation at the Third Polygraph Cache-Off [23]. The memory cache is 100MBytes (vs. 175MBytes used at the Cache-off). The *diskd* disk manager spins, waiting for request completion, if a daemon request queue is longer than 4K items, and it starts dropping file open requests when the queue exceeds 2K items. When using data-stream splice, all invocations use *AutoRelease*, 64K max2cache, and 16K min2read.

5 Experimental Evaluation

5.1 Microbenchmark: User-level Connection Tracking

Two microbenchmarks are used to compare the performance of the proposed *uselect*, with the *select* and *poll* system calls, and with the */dev/epoll* event notification mechanism. The first microbenchmark evaluates the scalability with the number of active connections for a fixed file size, and the second microbenchmark evaluates the impact of file size for a fixed rate. For both microbenchmarks, the hit ratio is almost 100%,

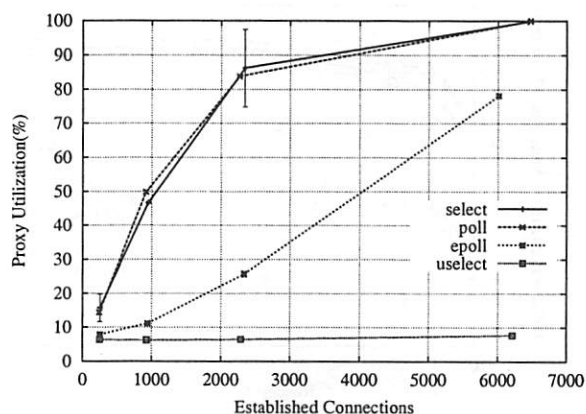


Figure 4: Proxy CPU utilization: 100%Hits, 100req/s, 8KByte files, null store.

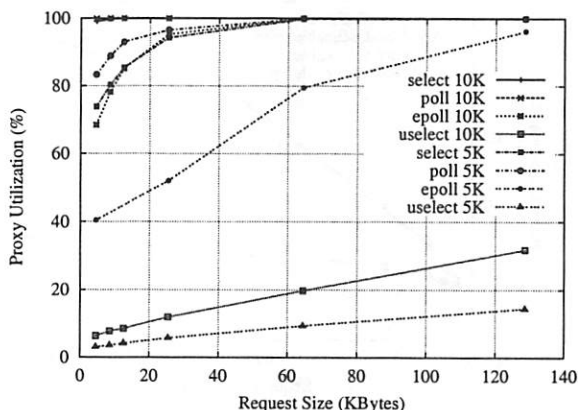


Figure 5: Proxy CPU utilization: 100%Hits, 5K robots at 50req/s and 10K robots at 100 req/s, null store.

which emulates the behavior of an origin Web site.

For the first microbenchmark, the Polygraph clients maintain a rate of 100 req/s, and a variable number of robots: 250, 1000, 2500 and 10,000. The file size is fixed at 8KBytes, which represents the ≈ 75 -th percentile of the Polymix-4 object size distribution. Figure 4 presents the proxy CPU utilization versus the mean number of concurrent established connections as reported by Polygraph, which may be lower than the total number of robots. The plots illustrate that, for this level of request rate, with *uselect*, the system load is independent of the number of active connections, while with */dev/epoll* and *select*, the system loads are very sensitive to the number of active connections. For instance, the load difference between 1000 and 2500 connections is 0% for *uselect*, 14% for *dev/epoll*, and 55% for *select*.

For the second microbenchmark, the Polygraph clients maintain 5,000 and 10,000 robots, each with a fixed rate of 0.01 req/s, resulting in overall request rates of 50 req/s and 100 req/s, respectively. File size varies

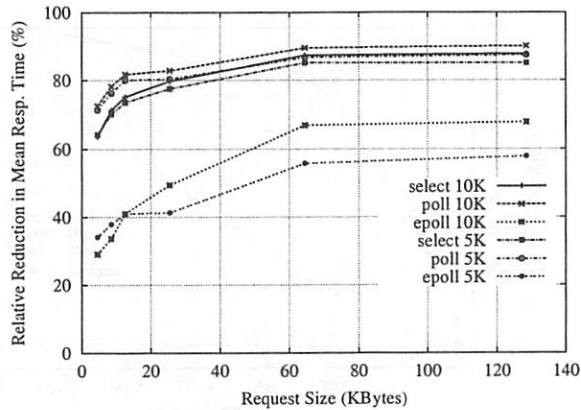


Figure 6: Rel.reduction of response time w/ uselect: 100%Hits, null store, 5K robots and 10K robots.

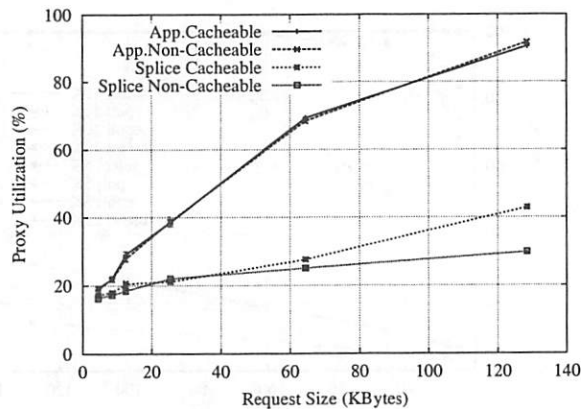


Figure 7: Proxy CPU utilization: 0%Hits, 40 req/s, null store.

from 4 to 128KBytes. We could not experiment with significantly more than 10K robots because of memory limitations on the proxy.

Figure 5 presents the variation of proxy CPU utilization. The plots illustrate that `uselect` is significantly more scalable than the other three mechanisms. For 5K active connections, the relative reduction in CPU utilization is 85-96%. For the larger load, the relative reduction is 68-94%. Moreover, the plot illustrates that `uselect` can handle 10K connections with lower overheads than `/dev/epoll` can handle 5K connections. The plot illustrates also that `/dev/epoll` is more scalable than `select` and `poll`, and that, at this loads, `select` is slightly more scalable than `poll`.

The lower CPU overheads achieved with `uselect` translate in significant response time reductions. Figure 6 presents the relative reductions computed as $100(1 - A.r/B.r)$, where $X.r$ is mean response time measured for configuration X . For 5K connections, the reduction relative to `/dev/epoll` is 29-58%, relative to `select` is 62-85%, and relative to `poll` is 70-85%.

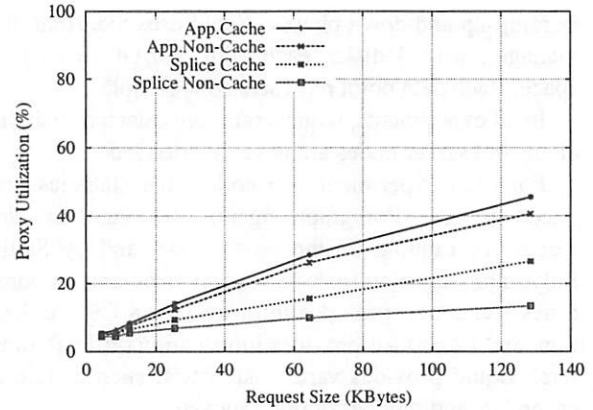


Figure 8: uselect proxy CPU utilization: 0%Hits, 40 req/s, null store.

5.2 Microbenchmark: Data-stream Splicing

Two microbenchmarks are used to evaluate the performance of data-stream splicing relative to application-level forwarding. The first microbenchmark evaluates the dependence on transfer lengths, and the second microbenchmark evaluates the dependence on system load.

For the first microbenchmark, Polygraph clients maintain a fixed rate of 40 req/s and a hit ratio of 0% (i.e., all requests handled by Squid require transfers from the server). Across experiments, we vary the object size and cacheability (i.e., cacheable or non-cacheable). When the object is cacheable, the splice mode is `KeepCopy`, thus the application performs read operations to bring the spliced content in its cache; no reads are performed for non-cacheable objects.

Figure 7 presents the proxy CPU utilization when the request rate is fixed at 40 req/s. This rate level is chosen to avoid reaching overload on both proxy and network. The plot illustrates that socket-level splice can result in significant overhead reductions. These reductions increase with the object size. Also, for large objects, reductions are larger for non-cacheable than for cacheable objects, for which `KeepCopy` is used. This is due to the fewer system calls executed for serving non-cacheable objects, difference which is relevant only for large objects. For non-cacheable objects, the relative reduction varies from 15% for 4K files, to 68% for 64K and 128K files. For cacheable objects, the relative reduction is 10-60%. For these experiments, the reduction in response time is relatively insignificant (up to 1.7%) because the mean response times is large (2.7-3.4s) due to server think time.

The performance benefit of data-stream splicing remains relevant also when using `uselect`. Figure 8 presents the CPU utilization when the application uses `uselect` instead of `select` to handle the same

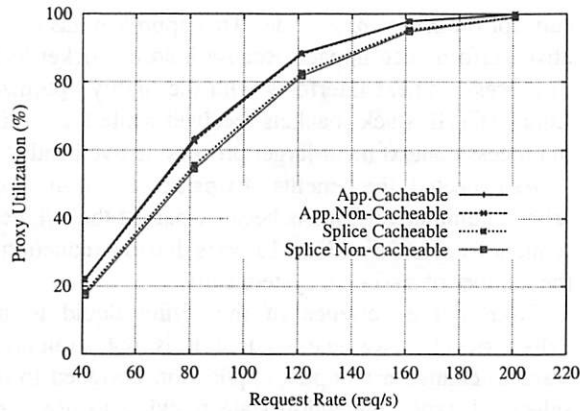


Figure 9: Proxy CPU utilization: 0% Hits, 8KByte files, null store.

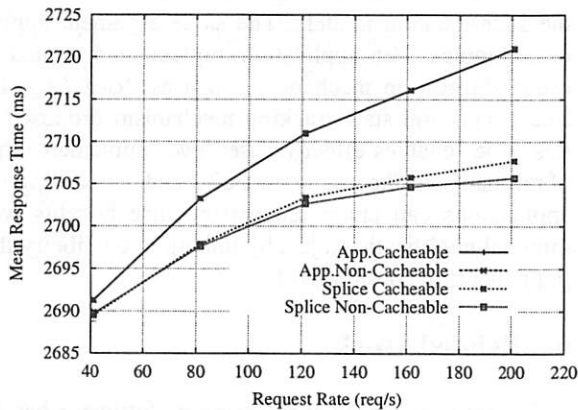


Figure 10: Mean Response Time: 0% Hits, 8KByte files, null store.

workload. The reduction relative to `uselect` with application-level forwarding is up to 45% for cacheable objects, and up to 65% for non-cacheable objects.

These experiments illustrate that data-stream splicing is a necessary mechanism in the context of the heavy-tail distribution of Web content sizes, because it helps Web proxies to significantly reduce the performance perturbations caused by serving atypically large files.

For the second microbenchmark, Polygraph clients generate request rates between 40 and 200 req/s and the file size is fixed at 8KBytes.

Figures 9 and 10 present the proxy CPU utilization and the mean response time, respectively; note that the plots for application-level forwarding overlap. These plots demonstrate that the reduction in CPU overhead enabled by splice increases with the request rate. However, as the system approaches overload, this reduction has a smaller impact on CPU utilization, but a larger impact on response time. For instance, for non-cacheable objects, the difference in CPU utilization decreases from 8% (at 120 req/s) to 0.9% (at 200 req/s), while the response time

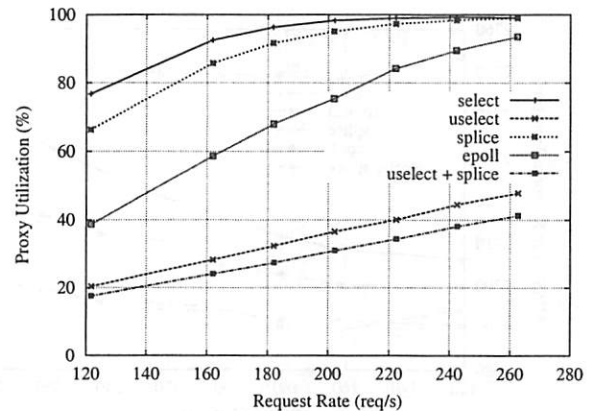


Figure 11: Proxy CPU utilization: Polymix-4, diskd store.

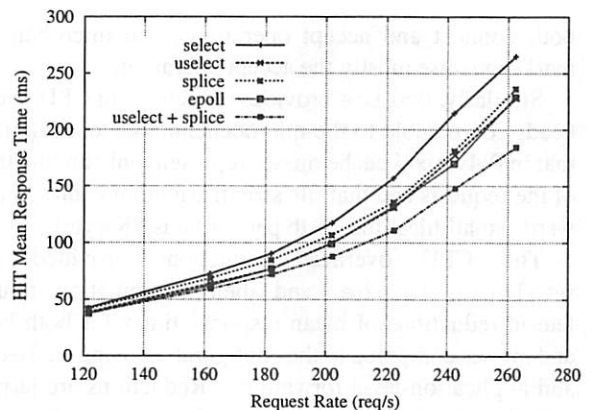


Figure 12: Hit Mean Response Time: Polymix-4, diskd store.

difference increases from 2ms to 14ms.

5.3 Polymix-4 Workload

In order to estimate the benefits of the proposed mechanisms in a real Web proxy deployment, we experiment with Polymix-4, a workload that provides a realistic mix of file sizes, cacheability types, and HTTP request types; it generates a realistic hit ratio, connection persistency model, and server think-time. This experiment evaluates `uselect`, `select`, `/dev/epoll` with application-level forwarding, and splicing with `select` and with `uselect`. Request rates vary from 120 to 260; swapping impacts the results at higher rates.

Figure 11 presents the proxy CPU utilization and Figures 12 and 13 present the mean response times for hits and misses, respectively. `uselect` provides reductions in CPU utilization similar to those in the microbenchmarks, 50-70% relative to `select` and 50% relative to `/dev/epoll`. These experiments also demonstrate that the `uselect` implementation can handle effectively

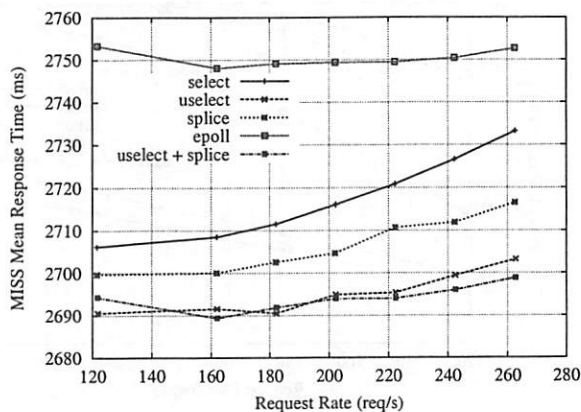


Figure 13: Miss Mean Response Time: Polymix-4, disk store.

both connect and accept operations; the microbenchmarks exercise mostly the accept operation.

Similarly, `splice` provides reductions in CPU overheads comparable to the microbenchmarks, considering that in Polymix-4 cache misses represent only about 40% of the requests and that file size distribution is biased towards small files (the 75-th percentile is 8KByte).

The CPU overhead reductions provided by `uselect`, `splice`, and their combination translate in reductions of mean response times for both hits and misses compared to the configuration using `select` and application-level forwarding. Reductions are larger at higher request rates. More specifically, the reductions for hits with `uselect` are 4-35ms, with `splice` are 1-28ms. Using both mechanisms, the reductions are 4-80ms, a 10-30% relative improvement. The reductions for miss with `uselect` are 15-30ms, and with `splice` are 6-16ms. Using both mechanisms yields reductions similar to `uselect` alone, which is a 0.5-1.3% relative improvement. We note that `/dev/epoll` provides reductions for hit response times similar to `uselect`, but the miss response times are the highest among the tested configurations, 20-50ms more than with `select`.

5.4 Discussions

Overall, the experimental evaluation presented in this paper illustrates that both user-level connection tracking and data-stream splice help lower the overheads and improve the scalability of Web servers. The former mechanism benefits any Web server that handles a very large numbers of concurrent connections, while the latter mechanism is mostly limited to Web proxy caches.

One lesson that we learned while experimenting with data-stream splice is that it is detrimental to forward data packets in process context. Our earlier implementations attempted to forward as many packets as possible in the `Splice` call, and upon the return from the `read` system

call, for the *KeepCopy* mode. This approach has a negative performance impact because holding socket locks in process context interferes with the highly optimized Linux TCP/IP stack: packets received while forwarding in process context incur larger processing overheads.

As expected, the benefits of `uselect` and `splice` are not cumulative. This is because part of the overhead reduction achieved with `splice` is due to a reduction in the number of `select` system calls.

From our experience of modifying Squid to use `/dev/epoll`, we learned that it is not straightforward to change a complex application designed to use `select/poll`-type connection state tracking to use event notification-based mechanisms, like `/dev/epoll`. It can be very difficult to identify all the code regions in the application built on programmer's assumptions about the state-tracking model. The same argument applies to a complex Web application implemented around an event notification mechanism, such as `/dev/epoll`. The connection state tracking mechanism proposed in this paper enables effective user-level implementations of mechanisms like `select/poll` and `/dev/epoll`; applications can enjoy the performance benefits with minimal modifications, just by linking to the library that implements the desired API.

6 Related Work

Recent research on Web server performance has focused on optimizing the operating system functions on the critical path of request processing. In this paper, we focus on the same problem domain, proposing two mechanisms that Web servers, and in particular Web proxy servers, can use in a flexible manner to reduce connection handling and data forwarding overheads. Both mechanisms address the overheads of context switching and data copy between application and kernel domains, one for connection state tracking and the other for data forwarding in TCP streams.

A large body of research has focused on improving the scalability of connection-state tracking mechanisms, critical for event-driven architectures. Traditional mechanisms for connection-state tracking, `select` and `poll` exhibit poor scalability. Optimizations can reduce the in-kernel overhead of collecting socket status information [4], but cannot reduce context switching and data copy overheads. Event delivery mechanisms with batch notifications represent an alternative to `select/poll` [5, 6, 19, 33, 34, 35]. This paradigm supports implementations that are more efficient, and reduces the overhead of data copy between user and kernel space, in particular when the number of active connections is a small fraction of the total number of open connections. The `/dev/epoll` proposed in [34] further reduces data

copy overheads by using a shared memory region between kernel and application for passing event notifications. Similarly, *ECalls* [33] uses shared memory for application-to-kernel and kernel-to-application notification. Overall, these proposals help reduce the system call overheads, but cannot reduce the number of invocations. Event delivery mechanisms with individual notifications, like I/O completion ports (IOCP) [12], incur a larger volume of system calls, but benefit a thread-based architecture by implementing a thread dispatching policy that minimizes thread context switches.

The connection state tracking mechanism introduced in this paper enables significant reductions of the number and overheads of system call invocations. By using connection state elements propagated by the kernel in a shared memory region, the application can acquire the information necessary for connection state tracking without context switching to the kernel domain. In the Linux implementation, connection state is propagated at user space automatically, after `connect` or the first `select/poll`; no system call other than `connect` and `accept` would be required if a file descriptor were assigned to the socket prior to the invocation of the `accept` handler. Existing APIs like `select`, event delivery [5, 34], `x` and IOCP [12] can be re-implemented to exploit the mechanism and achieve significant overhead reductions.

Numerous studies on TCP and server performance demonstrate that achievable transfer bandwidths are limited by the overhead of copying data between kernel and user-space buffers [7, 17]. Previous research has proposed several in-kernel splicing mechanisms of data streams produced by devices/files and sockets [11, 31]. In-kernel splicing of TCP connections has been proposed, as well. Some of the solutions [3, 9, 13, 15, 16] do not make the splicing interface available at application level. These solutions are integrated with kernel-level modules for HTTP request distribution and are implemented either between the TCP and socket layers [3, 15, 16] or in the IP layer [9, 13]. Existing solutions that can be exploited at application level are implemented in the IP layer [21, 22, 39] or in the socket layer [37], and are restricted in their ability to effectively serve the full range of connection characteristics and request types handled by a Web proxy. For instance, the IP-level implementations cannot handle pipelined requests and client aborts. The mechanism in [37] cannot handle cacheable content and persistent connections.

The data-stream splice mechanism proposed in this paper enables a Web proxy application to exploit kernel-level forwarding for all types of requests that involve transfers between its server and client connections. Similar to [37], the mechanism is implemented at socket level but with extended functionality. Drawing from the

socket level implementation, the mechanism has several advantages over the IP-level implementations. First, the mechanism allows the splicing of TCP connections with different maximum segment sizes or TCP options and fosters faster loss recovery [37]. Second, it allows for more efficient support for persistent connections (e.g., the mechanism in [39] unsplices at the first data received in the client connection) and for caching the transferred content (e.g., the mechanism in [22] aborts the caching procedure if the client aborts the connection). These advantages offset the relatively small increase in forwarding overheads vs. IP-level splicing due to the transport-layer processing on incoming and outgoing paths. We submit that IP-level solutions need to re-implement substantial segments of the TCP stack in order to support a flexible API, similar to the one proposed in this paper.

7 Conclusion

This paper proposes to enhance a general-purpose operating system with mechanisms that reduce the system overheads of applications such as Web servers, which handle large numbers of concurrent connections, and of applications such as Web proxies, which forward large volumes of data.

Improved scalability with the number of active connections is enabled by user-level connection tracking. Promoting a new implementation paradigm, this mechanism is the first to provide notifications of connection state changes without incurring any context switches and data copies between application and kernel domains. With a `select` API, this mechanism demonstrates CPU overhead reductions of 52-72% relative to `select` and 50% relative to `/dev/epoll`. In the future, we plan to implement an event notification API similar to [5].

Lower data forwarding overheads are enabled by data-stream splicing. Implemented in the socket layer, this mechanism is the first to enable effective in-kernel forwarding for the whole range of transfers performed by a Web proxy cache, supporting persistent connections, request pipelining, and content caching. Experiments demonstrate up to 12% reductions in Squid's forwarding overheads.

Acknowledgments. Authors thank John Tracey and Erich Nahum for providing the GigaBit equipment used in the evaluation. Also, authors thank Anees Shaikh for his comments on earlier versions of this paper.

References

- [1] T. Anderson, H. Levy, B. Bershad, E. Lazowska, "The Interaction of Architecture and Operating System Design", *ASPLOS*, 1991
- [2] Apache Software Foundation, "Apache http server

- project", <http://www.apache.org/>
- [3] H. Balakrishnan, V. Padmanabhan, S. Seshan, R. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links", *ACM SIGCOMM*, 1996
 - [4] G. Banga, J. Mogul, "Scalable kernel performance for Internet servers under realistic loads", *USENIX OSDI 1998*
 - [5] G. Banga, J. Mogul, P. Druschel, "A scalable and explicit event delivery mechanism for UNIX", *USENIX Annual Technical Conference*, 1999
 - [6] A. Chandra, D. Mosberger, "Scalability of Linux Event-Dispatch Mechanisms", *USENIX Annual Technical Conference*, 2001
 - [7] J. Chase, A. Gallatin, K. Yocum, "End-System Optimizations for High-Speed TCP", *IEEE Communications*, 39(4), Apr. 2001
 - [8] E. Cohen, H. Kaplan, J. Oldham, "Managing TCP connections under persistent HTTP", *World Wide Web Conference*, 1999
 - [9] A. Cohen, S. Rangarajan, H. Slye, "On the Performance of TCP Splicing for URL-aware Redirection", *USENIX Symposium on Internet Technologies and Systems*, 1999
 - [10] T. Dierks, C. Allen, "The TLS Protocol, Version 1.0", *IETF Network Working Group, RFC 2246*
 - [11] K. Fall, J. Pasquale, "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability", *USENIX Winter*, 1993
 - [12] J. Hart, "Win32 System Programming", *Addison Wesley Longman*, 1997
 - [13] G. Hunt, G. Goldszmidt, R. King, R. Mukherjee, "Network Dispatcher: A Connection Router for Scalable Internet Services", *World Wide Web Conference*, 1998
 - [14] S. Iyer, A. Rowstron, P. Druschel, "Squirrel: A decentralized peer-to-peer web cache", *ACM Symposium on Operating Systems Principles*, 2001
 - [15] IBM Corporation, "IBM Netfinity Web Server Accelerator V2.0", http://www.pc.ibm.com/us/solutions/netfinity/server_accelerator.html
 - [16] P. Joubert, R. King, R. Neves, M. Russinovich, J. Tracey, "High-Performance Memory-Based Web Servers: Kernel and User-Space Performance", *USENIX Annual Technical Conference*, 2001
 - [17] J. Kay, J. Pasquale, "Profiling and Reducing Processing Overheads in TCP/IP", *IEEE/ACM Transactions on Networking*, 4(6) p.817-828, 1996
 - [18] B. Krishnamurthy, C. Wills, "Improving Web Performance by Client Characterization Driven Server Adaptation", *World Wide Web Conference*, 2002
 - [19] J. Lemon, "Kqueue: A generic and scalable event notification facility", *FREENIX Track: USENIX Annual Technical Conference*, 2001
 - [20] D. Libenzi, "Improving (network) I/O performance", <http://www.xmailserver.org/linux-patches/nio-improve.html>
 - [21] D. Maltz, P. Bhagwat, "MSOCKS: An Architecture for Transport Layer Mobility", *INFOCOM*, 1998
 - [22] D. Maltz, P. Bhagwat, "Improving HTTP Caching Proxy Performance with TCP Tap", *IBM Research Report RC 21147*, Mar. 1998
 - [23] The Measurement Factory, "Public Benchmarking Results", <http://www.measurement-factory.com/results>
 - [24] E. Nahum, T. Barzilai, D. Kandlur, "Performance Issues in WWW Servers", *ACM SIGMETRICS*, 1999
 - [25] E. Nahum, M. Roşu, S. Seshan, J. Almeida, "The Effects of Wide Area Conditions on WWW Server Performance", *ASM SIGMETRICS*, 2001
 - [26] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, R. Tewari, "Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks", *World Wide Web Conference*, 2002
 - [27] NLNR, "Squid Web Proxy Cache", <http://www.squid-cache.org/>
 - [28] NLNR, "IRCaché Home", <http://www.ircache.net>
 - [29] J. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?", *Summer 1990 USENIX Conference*
 - [30] V. Pai, P. Druschel, W. Zwaenepoel, "Flash: An Efficient and Portable Web Server", *USENIX Annual Technical Conference*, 1999
 - [31] J. Pasquale, E. Anderson, K. Fall, J. Kay, "High-Performance I/O and Networking Software in Sequoia 2000", *Digital Technical Journal*, 1995
 - [32] V. Paxson, "End-to-end Internet packet dynamics", *IEEE/ACM Transactions on Networking*, 7(3), June 1999
 - [33] C. Poellabauer, K. Schwan, R. West, "Lightweight Kernel/User Communication for Real-Time and Multimedia Applications", *Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2001
 - [34] N. Provos, C. Lever, "Scalable network I/O in Linux", *Technical Report CITI-TR-00-4*, University of Michigan, Center for Information Technology, 2000
 - [35] N. Provos, C. Lever, S. Tweedie, "Analyzing the Overhead Behavior of a Simple Web Server", *Technical Report CITI-TR-00-7*, University of Michigan, Center for Information Technology, 2000
 - [36] L. Rizzo, "dummynet", <http://info.iet.unipi.it/~luigi/ip-dummynet>
 - [37] M. Rosu, D. Rosu, "An Evaluation of TCP Splice Benefits in Web Proxy Servers", *World Wide Web Conference*, 2002
 - [38] M. Rosu, D. Rosu, "Kernel Support for Faster Web Proxies", *IBM Research Report RC 22669*, Dec. 2002
 - [39] O. Spatscheck, J. Hansen, J. Hartman, L. Peterson, "Optimizing TCP Forwarder Performance", *IEEE/ACM Transactions on Networking*, 8(2), April 2000, also *Dept. of CS, Univ. of Arizona, TR 98-01, Feb. 1998*
 - [40] Web Polygraph, "Workloads", <http://www.web-polygraph.org>
 - [41] M. Welsh, D. Culler, E. Brewer, "SEDA: An Architecture for Well-Conditioned scalable Internet Services", *ACM Symposium on Operating Systems Principles*, 2001
 - [42] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, H. Levy, "On the scale and performance of cooperative Web proxy caching", *ACM Symposium on Operating Systems Principles*, 1999
 - [43] Zeus Technology Limited, "Zeus Web Server", <http://www.zeus.co.uk>

Multiprocessor Support for Event-Driven Programs

Nickolai Zeldovich*, Alexander Yip, Frank Dabek,
Robert T. Morris, David Mazières†, Frans Kaashoek

nickolai@cs.stanford.edu, {yipal, fdabek, rtm, kaashoek}@lcs.mit.edu, dm@cs.nyu.edu

MIT Laboratory for Computer Science

200 Technology Square

Cambridge, MA 02139

Abstract

This paper presents a new asynchronous programming library (*libasync-smp*) that allows event-driven applications to take advantage of multiprocessors by running code for event handlers in parallel. To control the concurrency between events, the programmer can specify a *color* for each event: events with the same color (the default case) are handled serially; events with different colors can be handled in parallel. The programmer can incrementally expose parallelism in existing event-driven applications by assigning different colors to computationally-intensive events that do not share mutable state.

An evaluation of *libasync-smp* demonstrates that applications achieve multiprocessor speedup with little programming effort. As an example, parallelizing the cryptography in the SFS file server required about 90 lines of changed code in two modules, out of a total of about 12,000 lines. Multiple clients were able to read large cached files from the *libasync-smp* SFS server running on a 4-CPU machine 2.5 times as fast as from an unmodified uniprocessor SFS server on one CPU. Applications without computationally intensive tasks also benefit: an event-driven Web server achieves 1.5 speedup on four CPUs with multiple clients reading small cached files.

1 Introduction

To obtain high performance, servers must overlap computation with I/O. Programs typically achieve this overlap using threads or events. Threaded programs typically process each request in a separate thread; when one thread blocks waiting for I/O, other threads can run. Threads provide an intuitive programming model and can take advantage of multiprocessors; however, they

require coordination of accesses by different threads to shared state, even on a uniprocessor. In contrast, event-based programs are structured as a collection of *callback* functions which a main loop calls as I/O events occur. Event-based programs execute callbacks serially, so the programmer need not worry about concurrency control; however, event-based programs until now have been unable to take full advantage of multiprocessors without running multiple copies of an application or introducing fine-grained synchronization.

The contribution of this paper is *libasync-smp*, a library that supports event-driven programs on multiprocessors. *libasync-smp* is intended to support the construction of user-level systems programs, particularly network servers and clients; we show that these applications can achieve performance gains on multiprocessors by exploiting coarse-grained parallelism. *libasync-smp* is intended for programs that have natural opportunities for parallel speedup; it has no support for expressing very fine-grained parallelism. The goal of *libasync-smp*'s concurrency control mechanisms is to provide enough concurrency to extract parallel speedup without requiring the programmer to reason about the correctness of a fine-grained parallel program.

Much of the effort required to make existing event-driven programs take advantage of multiprocessors is in specifying which events may be handled in parallel. *libasync-smp* provides a simple mechanism to allow the programmer to incrementally add parallelism to uniprocessor applications as an optimization. This mechanism allows the programmer to assign a *color* to each callback. Callbacks with different colors can execute in parallel. Callbacks with the same color execute serially. By default, *libasync-smp* assigns all callbacks the same color, so existing programs continue to work correctly without modification. As programmers discover opportunities to safely execute callbacks in parallel, they can assign different colors to those callbacks.

libasync-smp is based on the *libasync* library [16].

*Stanford University

†New York University

libasync uses operating system asynchronous I/O facilities to support event-based programs on uniprocessors. The modifications for *libasync-smp* include coordinating access to the shared internal state of a few *libasync* modules, adding support for colors, and scheduling callbacks on multiple CPUs.

An evaluation of *libasync-smp* demonstrates that applications achieve multiprocessor speedup with little programming effort. As an example, we modified the SFS [17] file server to use *libasync-smp*. This server uses more than 260 distinct callbacks. Most of the CPU time is spent in just two callbacks, those responsible for encrypting and decrypting client traffic; this meant that coloring just a few callbacks was sufficient to gain substantial parallel speedup. The changes affected 90 lines in two modules, out of a total of about 12,000 lines. When run on a machine with four Intel Xeon CPUs, the modified SFS server was able to serve large cached files to multiple clients 2.5 times as fast as an unmodified uniprocessor SFS server on one CPU.

Even servers without CPU-intensive operations such as cryptography can achieve speedup approaching that offered by the operating system, especially if the O/S kernel can take advantage of a multiprocessor. For example, with a workload of multiple clients reading small cached files, an event-driven web server achieves 1.5 speedup on four CPUs.

The next section (Section 2) introduces *libasync*, on which *libasync-smp* is based, and describes its support for uniprocessor event-driven programs. Section 3 and 4 describe the design and implementation of *libasync-smp*, and show examples of how applications use it. Section 5 uses two examples to show that use of *libasync-smp* requires little effort to achieve parallel speedup. Section 6 discusses related work, and Section 7 concludes.

2 Uniprocessor Event-driven Design

Many applications use an event-driven architecture to overlap slow I/O operations with computation. Input from outside the program arrives in the form of events; events can indicate, for example, the arrival of network data, a new client connection, completion of disk I/O, or a mouse click. The programmer structures the program as a set of callback functions, and registers interest in each type of event by associating a callback with that event type.

In the case of complex event-driven servers, such as named [7], the complete processing of a client request may involve a sequence of callbacks; each consumes an event, initiates some I/O (perhaps by sending a request

packet), and registers a further callback to handle completion of that particular I/O operation (perhaps the arrival of a specific response packet). The event-driven architecture allows the server to keep state for many concurrent I/O activities.

Event-driven programs typically use a library to support the management of events. Such a library maintains a table associating incoming events with callbacks. The library typically contains the main control loop of the program, which alternates between waiting for events and calling the relevant callbacks. Use of a common library allows callbacks from mutually ignorant modules to co-exist in a single program.

An event-driven library's control loop typically calls ready callbacks one at a time. The fact that the callbacks never execute concurrently simplifies their design. However, it also means that an event-driven program typically cannot take advantage of a multiprocessor.

The multiprocessor event-driven library described in this paper is based on the *libasync* uniprocessor library originally developed as part of SFS [17, 16]. This section describes uniprocessor *libasync* and the programming style involved in using it. Existing systems, such as named [7] and Flash [19], use event-dispatch mechanisms similar to the one described here. The purpose of this section is to lay the foundations for Section 3's description of extensions for multiprocessors.

2.1 *libasync*

libasync is a UNIX C++ library that provides both an event dispatch mechanism and a collection of event-based utility modules for functions such as DNS host name lookup and Sun RPC request/reply dispatch [16]. Applications and utility modules register callbacks with the *libasync* dispatcher. *libasync* provides a single main loop which waits for new events with the UNIX `select()` system call. When an event occurs, the main loop calls the corresponding registered callback. Multiple modules can use *libasync* without knowing about each other, which encourages modular design and reusable code.

libasync handles a core set of events as well as a set of events implemented by utility modules. The core events include new connection requests, the arrival of data on file descriptors, timer expiration, and UNIX signals. The RPC utility module allows automatic parsing of incoming Sun RPC calls; callbacks registered per program/procedure pair are invoked when an RPC arrives. The RPC module also allows a callback to be registered to handle the arrival of the reply to a particular RPC call. The DNS module supports non-blocking concurrent host name lookups. Finally, a file I/O module allows applica-

tions to perform non-blocking file system operations by sending RPCs to the NFS server in the local kernel; this allows non-blocking access to all file system operations, including (for example) file name lookup.

Typical programs based on *libasync* register a callback at every point at which an equivalent single-threaded sequential program might block waiting for input. The result is that programs create callbacks at many points in the code. For example, the SFS server creates callbacks at about 100 points.

In order to make callback creation easy, *libasync* provides a type-checked facility similar to function-currying [23] in the form of the `wrap()` macro [16]. `wrap()` takes a function and values as arguments and returns an anonymous function called a *wrap*. If `w = wrap(fn, x, y)`, for example, then a subsequent call `w(z)` will result in a call to `fn(x, y, z)`. A *wrap* can be called more than once; *libasync* reference-counts *wraps* and automatically frees them in order to save applications tedious book keeping. Similarly, the library also provides support for programmers to pass reference-counted arguments to *wrap*. The benefit of `wrap()` is that it simplifies the creation of callback structures that carry state.

2.2 Event-driven Programming

Figure 1 shows an abbreviated fragment of a program written using *libasync*. The purpose of the application is to act as a web proxy. The example code accepts TCP connections, reads an HTTP request from each new connection, extracts the server name from the request, connects to the indicated server, etc. One way to view the example code is that it is the result of writing a single sequential function with all these steps, and then splitting it into callbacks at each point that the function would block for input.

`main()` calls `inetsocket()` to create a socket that listens for new connections on TCP port 80. UNIX makes such a socket appear readable when new connections arrive, so `main()` calls the *libasync* function `fdcb()` to register a read callback. Finally `main()` calls `amain()` to enter the *libasync* main loop.

The *libasync* main loop will call the callback `wrap` with no arguments when a new connection arrives on `afd`. The `wrap` calls `accept_cb()` with the other arguments passed to `wrap()`, in this case the file descriptor `afd`. After allocating a buffer in which to accumulate client input, `accept_cb()` registers a callback to `req_cb()` to read input from the new connection. The server keeps track of its state for the connection, which consists of the file descriptor and the buffer, by including it in each `wrap()` call and thus passing it from one

```
main()
{
    // listen on TCP port 80
    int afd = inetsocket(SOCK_STREAM, 80);
    // register callback for new connections
    fdcb(afd, READ, wrap(accept_cb, afd));
    amain(); // start main loop
}

// called when a new connection arrives
accept_cb(int afd)
{
    int fd = accept(afd, ...);
    str inBuf(""); // new ref-counted buffer
    // register callback for incoming data
    fdcb(fd, READ, wrap(req_cb, fd, inBuf));
}

// called when data arrives
req_cb(int fd, str inBuf)
{
    read(fd, buf, ...);
    append input to inBuf;
    if(complete request in inBuf){
        // un-register callback
        fdcb(fd, READ, NULL);

        // parse the HTTP request
        parse_request(inBuf, serverName, file);

        // resolve serverName and connect
        // both are asynchronous
        tcpconnect(serverName, 80,
                   wrap(connect_cb, fd, file));
    } else {
        // do nothing; wait for more calls to req_cb()
    }
}

// called when we have connected to the server
connect_cb(int client_fd, str file, int server_fd)
{
    // write the request when the socket is ready
    fdcb(server_fd, WRITE,
          wrap(write_cb, file, server_fd));
}
```

Figure 1: Outline of a web proxy that uses *libasync*.

callback to the next. If multiple clients connect to the proxy, the result will be multiple callbacks waiting for input from the client connections.

When a complete request has arrived, the proxy server needs to look up the target web server's DNS host name and connect to it. The function `tcpconnect()` performs both of these tasks. The DNS lookup itself involves waiting for a response from a DNS server, perhaps more than one in the case of timeouts; thus the *libasync* DNS resolver is internally structured as a set of callbacks. Waiting for TCP connection establishment to complete also involves callbacks. For these reasons, `tcpconnect()` takes a *wrap* as one of its arguments, carries that *wrap* along in its own callbacks, and finally calls the *wrap* when the connection process completes or fails. This style of programming is reminiscent of the continuation-passing style [21], and makes it easy for programmers to compose modules.

A number of applications are based on *libasync*; Figure 2 lists some of them, along with the number of distinct calls to `wrap()` in each program. These numbers give a feel for the level of complexity in the programs' use of callbacks.

Name	#Wraps	Lines of Code
SFS [17]	229	39871
SFSRO [13]	58	4836
Chord [22]	65	5445
CFS [10]	87	4960

Figure 2: Applications based on *libasync*, along with the approximate number of distinct calls to `wrap()` in each application. The numbers are exclusive of the wraps created by *libasync* itself, which number about 30.

2.3 Interaction with multiprocessors

A single event-driven process derives no direct benefit from a multi-processor. There may be an indirect speedup if the operating system or helper processes can make use of the multiprocessor's other CPUs.

It is common practice to run multiple independent copies of an event-driven program on a multiprocessor. This *N-copy* approach might work in the case of a web server, since the processing of different client requests can be made independent. The *N-copy* approach does not work if the program maintains mutable state that is shared among multiple clients or requests. For example, a user-level file server might maintain a table of leases for client cache consistency. In other cases, running multiple independent copies of a server may lead to a decrease in efficiency. A web proxy might maintain a cache of recently accessed pages: multiple copies of the proxy could maintain independent caches, but content duplicated in these caches would waste memory.

3 Multiprocessor Design

The focus of this paper is *libasync-smp*, a multiprocessor extension of *libasync*. The goal of *libasync-smp* is to execute event-driven programs faster by running callbacks on multiple CPUs. Much of the design of *libasync-smp* is motivated by the desire to make it easy to adapt existing *libasync*-based servers to multiprocessors. The goal of the *libasync-smp* design is to allow both the parallelism of the *N-copy* arrangement and the advantages of shared data structures.

A server based on *libasync-smp* consists of a single process containing one worker thread per available CPU. Each thread repeatedly chooses a callback from a set of runnable callbacks and runs it. The threads share an address space, file descriptors, and signals. The library assumes that the number of CPUs available to the process is static over its running time. A mechanism such as sched-

uler activations [2] could be used to dynamically determine the number of available CPUs.

There are a number of design challenges to making the single address space approach work, the most interesting of which is coordination of access to application data shared by multiple callbacks. An effective concurrency control mechanism should allow the programmer to easily (and incrementally) identify which parts of a server can safely be run in parallel.

3.1 Coordinating callbacks

The design of the concurrency control mechanisms in *libasync-smp* is motivated by two observations. First, system software often has natural coarse-grained parallelism, because different requests don't interact or because each request passes through a sequence of independent processing stages. Second, existing event-driven programs are already structured as non-blocking units of execution (callbacks), often associated with one stage of the processing for a particular client. Together, these observations suggest that individual callbacks are an appropriate unit of coordination of execution.

libasync-smp associates a *color* with each registered callback, and ensures that no two callbacks with the same color execute in parallel. Colors are arbitrary 32-bit values. Application code can optionally specify a color for each callback it creates; if it specifies no color, the callback has color zero. Thus, by default, callbacks execute sequentially on a single CPU. This means that unmodified event-driven applications written for *libasync* will execute correctly with *libasync-smp*.

The orthogonality of color to the callback's code eases the adaptation of existing *libasync*-based servers. A typical arrangement is to run the code that accepts new client connections in the default color. If the processing for different connections is largely independent, the programmer assigns each new connection a new unique color that applies to all the callbacks involved in processing that connection. If a particular stage in request processing shares mutable data among requests (e.g. a cache of web pages), the programmer chooses a color for that stage and applies it to all callbacks that use the shared data, regardless of which connection the callback is associated with.

In some cases, application code may need to be restructured to permit callbacks to be parallelized. For example, a single callback might use shared data but also have significant computation that does not use shared data. It may help to split such a callback; the first half would use a special *libasync-smp* call (`cpucb()`) to schedule the second half with a different color.

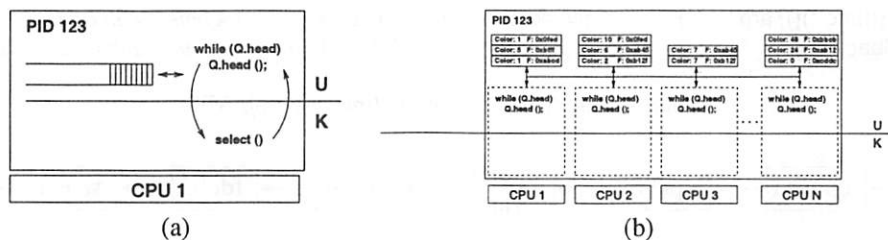


Figure 3: The single process event driven architecture (left) and the *libasync-smp* architecture (right). Note that in the *libasync-smp* architecture callbacks of the same color appear in the same queue. This guarantees that callbacks with the same color are never run in parallel and always run in the order in which they were scheduled.

The color mechanism is less expressive than locking; for example, a callback can have only one color, which is equivalent to holding a single lock for the complete duration of a callback. However, experience suggests that fine-grained and sophisticated locking, while it may be necessary for correctness with concurrent threads, is rarely necessary to achieve reasonable speedup on multiple CPUs for server applications. Parallel speedup usually comes from the parts of the code that don't need much locking; coloring allows this speedup to be easily captured, and also makes it easy to port existing event-driven code to multiprocessors.

3.2 *libasync-smp* API

The API that *libasync-smp* presents differs slightly from that exposed by *libasync*. The `cwrap()` function is analogous to the `wrap()` function described in Section 2 but takes an optional color argument; Table 1 shows the `cwrap()` interface. The color specified at the callback's creation (i.e. when `cwrap()` is called) dictates the color it will be executed under. Embedding color information in the callback object rather than in an argument to `fdcb()` (and other calls which register callbacks) allows the programmer to write modular functions which accept callbacks and remain agnostic to the color under which those callbacks will be executed. Note that colors are not inherited by new callbacks created inside a callback running under a non-zero color. While color inheritance might seem convenient, it makes it very difficult to write modular code as colors "leak" into modules which assume that callbacks they create carry color zero.

Since colors are arbitrary 32-bit values, programmers have considerable latitude in how to assign colors. One reasonable convention is to use each request's file descriptor number as the color for its parallelizable callbacks. Another possibility is to use the address of a data structure to which access must be serialized; for example, a per-client or per-request state structure. Depending on the convention, it could be the case that unrelated

modules accidentally choose the same color. This might reduce performance, but not correctness.

libasync-smp provides a `cpucb()` function that schedules a callback for execution as soon as a CPU is idle. The `cpucb()` function can be used to register a callback with a color different from that of the currently executing callback. A common use of `cpucb()` is to split a CPU-intensive callback in two callbacks with different colors, one to perform a computation and the other to synchronize with shared state. To minimize programming errors associated with splitting an existing callback into a chain of `cpucb()` callbacks, *libasync-smp* guarantees that all CPU callbacks of the same color will be executed in the order they were scheduled. This maintains assumptions about sequential execution that the original single callback may have been relying on. Execution order isn't defined for callbacks with different colors.

3.3 Example

Consider the web proxy example from Section 2. For illustrative purposes assume that the `parse_request()` routine uses a large amount of CPU time and does not depend on any shared data. We could re-write `req_cb()` to parse different requests in parallel on different CPUs by calling `cpucb()` and assigning the callback a unique color. Figure 4 shows this change to `req_cb()`. In this example only the `parse_request()` workload is distributed across CPUs. As a further optimization, reading requests could be parallelized by creating the read request callback using `cwrap()` and specifying the request's file descriptor as the callback's color.

3.4 Scheduling callbacks

Scheduling callbacks involves two operations: placing callbacks on a worker thread's queue and, at each thread, deciding which callback to run next.


```
callback cwrap ((func *)(), arg1, arg2, ..., argN, color c = 0) // Create a callback object with color c.
void cpucb (callback cb) // Add cb to the runnable callback queue immediately.
```

Table 1: Sample calls from the *libasync-smp* API.



Figure 5: The callback queue structure in *libasync-smp*. `cpucb()` adds new callbacks to the left of the dummy element marked “cpucb Tail.” New I/O callbacks are added at “Queue Tail.” The scheduler looks for work starting at “Queue Head.”

```
// called when data arrives
req_cb(int fd, str inBuf)
{
    read(fd, buf, ...);
    append input to inBuf;
    if(complete request in inBuf){
        // un-register callback
        fdcb(fd, READ, NULL);

        // parse the HTTP request under color fd
        cpucb (cwrap (parse_request_cb, fd, inBuf,
                    (color)fd))
    } else {
        // do nothing; wait for
        // more calls to req_cb()
    }
}

// below parsing done w/ color fd
parse_req_cb (int fd, str inBuf)
{
    parse_request (inBuf, serverName, file);

    // start connection to server
    tcpconnect (serverName, wrap(connect_cb, fd, file));
}
```

Figure 4: Changes to the asynchronous web proxy to take advantage of multiple CPUs

A callback is placed on a thread’s queue in one of two ways: due to a call to `cpucb()` or because the *libasync-smp* main loop detected the arrival of an I/O, timer, or signal event for which a callback had been registered. A callback with color c is placed in the queue of worker thread $c \bmod N$ where N is the number of worker threads. This simple rule distributes callbacks approximately evenly among the worker threads. It also preserves the order of activation of callbacks with the same color and may improve cache locality.

If a worker thread’s task queue is empty it attempts to steal work from another thread’s queue [9]. Work must be stolen at the granularity of all callbacks of the same color and the color to be stolen must not be executing currently to preserve guarantees on ordering of callbacks within the same color. *libasync-smp* consults a per-thread field containing the currently running color to guarantee the latter requirement.

When a color is moved from one thread to another, future callbacks of that color will be assigned to the new queue; otherwise, callbacks of the same color might execute in parallel. To ensure that all callbacks with the

same color appear on the same queue, the library maintains a mapping of colors to threads: the n th element of a 1024 element array indicates which thread should execute all colors which are congruent to $n \pmod{1024}$. This array is initialized in such a way as to give the initial distribution described above.

Each *libasync-smp* worker thread uses a simple scheduler to choose a callback to execute next from its queue. The scheduler considers priority and callback/thread affinity when choosing colors; its design is loosely based on that of the Linux SMP kernel [8].

The scheduler favors callbacks of the same color as the last callback executed by the worker in order to increase performance. Callback colors often correspond to particular requests, so *libasync-smp* tends to run callbacks from the same request on the same CPU. This processor-callback affinity leads to greater cache hit rates and improved performance.

When *libasync-smp* starts, it adds a “select callback” to the run queue of the worker thread responsible for color zero. This callback calls `select()` to detect I/O events. The select callback enqueues callbacks in the appropriate queue based on which file descriptors `select()` indicates have become ready.

The select callback might block the worker thread that calls it if no file descriptors are ready; this would prevent one CPU from executing any other tasks in its work queue. To avoid this, the select callback uses `select()` to poll without blocking. If `select()` returns some file descriptors, the select callback adds callbacks for those descriptors to the work queue, and then puts itself back on the queue. If no file descriptors were returned, a *blocking* select callback is placed back on the queue instead. The blocking select callback is only run if it is the only callback on the queue, and calls `select()` with a non-zero timeout. In all other aspects, it behaves just like the non-blocking select callback.

The use of the two select callbacks along with work stealing guarantees that a worker thread never blocks in `select()` when there are callbacks eligible to be exe-

cuted in the system.

Figure 5 shows the structure of a queue of runnable callbacks. In general, new runnable callbacks are added on the right, but `cpucb()` callbacks always appear to the left of I/O event callbacks. A worker thread's scheduler considers callbacks starting at the left. The scheduler examines the first few callbacks on the queue. If among these callbacks the scheduler finds a callback whose color is the same as the last callback executed on the worker thread, the scheduler runs that callback. Otherwise the scheduler runs the left-most eligible callback.

The scheduler favors `cpucb()` callbacks in order to increase the performance of chains of `cpucb()` callbacks from the same client request. The state used by a `cpucb()` callback is likely to be in cache because the creator of the `cpucb()` callback executed recently. Thus, early execution of `cpucb()` callbacks increases cache locality.

4 Implementation

libasync-smp is an extension of *libasync*, the asynchronous library [16] distributed as part of the SFS file system [17]. The library runs on Linux, FreeBSD and Solaris. Applications written for *libasync* work without modification with *libasync-smp*.

The worker threads used by *libasync-smp* to execute callbacks are kernel threads created by a call to the `clone()` system call (under Linux), `rfork()` (under FreeBSD) or `thr_create()` (under Solaris).

Although programs which use *libasync-smp* should not need to perform fine grained locking, the *libasync-smp* implementation uses spin-locks internally to protect its own data structures. The most important locks protect the callback run queues, the callback registration tables, retransmit timers in the RPC machinery, and the memory allocator.

The source code for *libasync-smp* is available as part of the SFS distribution at <http://www.fs.net> on the CVS branch `mp-async`.

5 Evaluation

In evaluating *libasync-smp* we are interested in both its performance and its usability. This section evaluates the parallel speedup achieved by two sample applications using *libasync-smp*, and compares it to the speedup achieved by existing similar applications. We also evaluate usability in terms of the amount of programmer effort required to modify existing event-driven programs to get

good parallel speedup.

The two sample applications are the SFS file server and a caching web server. SFS is an ideal candidate for achieving parallel speedup using *libasync-smp*: it is written using *libasync* and performs compute intensive cryptographic tasks. Additionally, the SFS server maintains state that can not be replicated among independent copies of the server. A web server is a less promising candidate: web servers do little computation and all state maintained by the server can be safely shared. Accordingly we expect good SMP speedup from the SFS server and a modest improvement in performance from the web server.

All tests were performed on a SMP server equipped with four 500 MHz Pentium III Xeon processors. Each processor has 512KB of cache and the system has 512MB of main memory. The disk subsystem consists of a single ultra-wide 10,000 RPM SCSI disk. Load was generated by four fast PCs running Linux, each connected to the server via a dedicated full-duplex gigabit Ethernet link. Processor scaling results were obtained by completely disabling all but a certain number of processors on the server.

The server runs a slightly modified version of Linux kernel 2.4.18. The modification removes a limit of 128 on the number of new TCP connections the kernel will queue awaiting an application's call to `accept()`. This limit would have prevented good server performance with large numbers of concurrent TCP clients.

5.1 HTTP server

To explore whether we can use *libasync-smp* to achieve multiprocessor speedup in applications where the majority of computation is not concentrated in a small portion of the code, we measured the performance of an event-driven HTTP 1.1 web server.

The web server uses an NFS loop-back server to perform non-blocking disk I/O. The server process maintains two caches in its memory: a web page cache and a file handle cache. The former holds the contents of recently served web pages while the latter caches the NFS file handles of recently accessed files. The page cache is split into a small number (10) of independent caches to allow simultaneous access [6]. Both of the file handle cache and the individual page caches must be protected from simultaneous access.

5.1.1 Parallelizing the HTTP server

Figure 6 illustrates the concurrency present in the web server when it is serving concurrent requests for pages not in the cache. Each vertical set of circles represents a

single callback, and the arrows connect successive callbacks involved in processing a request. Callbacks that can execute in parallel for different requests are indicated by multiple circles. For instance, the callback that reads an HTTP request from the client can execute in parallel with any other callback. Other steps involve access to shared mutable data such as the page cache; callbacks must execute serially in these steps.

When the server accepts a new connection, it colors the callback that reads the connection's request with its file descriptor number. The callback that writes the response back to the client is similarly colored. The shared caches are protected by coloring all operations that access a given cache the same color. Only one callback may access each cache simultaneously; however, two callbacks may access two distinct caches simultaneously (i.e. one request can read a page cache while another reads the file handle cache). The code that sends RPCs to the loop-back NFS server to read files is also serialized using a single color. This was necessary since the underlying RPC machinery maintains state about pending RPCs which could not safely be shared. The state maintained by the RPC layer is a candidate for protection via internal mutexes; if this state were protected within the library the "read file" step could be parallelized in the web server.

While this coloring allows the caches and RPC layer to operate safely, it reveals a limitation of coloring as a concurrency control mechanism. Ideally, we should allow any number of callbacks to read the cache, but limit the number of callbacks accessing the cache to one if the cache is being written. This read/write notion is not expressible with the current locking primitives offered by *libasync-smp* although they could be extended to include it [4]. We did not implement read/write colors since dividing the page cache into smaller, independent caches provided much of the benefit of read/write locks without requiring modifications to the library.

The server also delegates computation to additional CPUs using calls to `cpucb()`. When parsing a request the server looks up the longest match for the pathname in the file handle cache (which is implemented as a hash table). To move the computation of the hash function out of the cache color, we use a `cpucb()` callback to first hash each prefix of the path name, and then, in a callback running as the cache color, search for each hash value in the file handle cache.

In all, 23 callbacks were modified to include a color argument or to be invoked via a `cpucb()` (or both). The web server has 1,260 lines of code in total, and 39 calls to wrap.

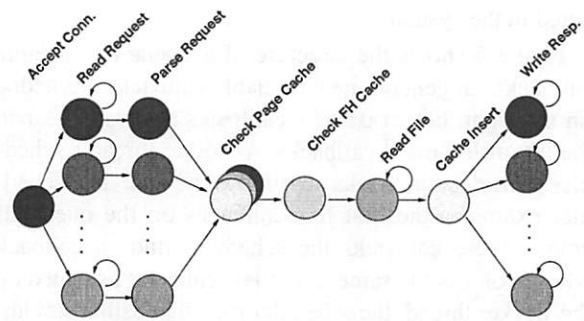


Figure 6: The sequence of callbacks executed when the *libasync-smp* web server handles a request for a page not in the cache. Nodes represent callbacks, arrows indicate that the node at the source scheduled the callback represented by the node at the tip. Nodes on the same vertical line are run under distinct colors (and thus potentially in parallel). The stacked circles in the "Check page cache" stage indicate that a small number of threads (less than the number of concurrent requests) can access the cache simultaneously). Labels at the top of the figure describe each step of the processing.

5.1.2 HTTP server performance

To demonstrate that the web server can take advantage of multiprocessor hardware, we tested the performance of the parallelized web server on a cache-based workload while varying the number of CPUs available to the server. The workload consisted of 720 files whose sizes were distributed according to the SPECweb99 benchmark [20]; the total size of the data set was 100MB which fits completely into the server's in-memory page cache. Four machines simulated a total of 800 concurrent clients. A single instance of the load generation client is capable of reading over 20MB/s from the web server. Each client made 10 requests over a persistent connection before closing the connection and opening a new one. The servers were started with cold caches and run for 4 minutes under load. The server's throughput was then measured for 60 seconds, to capture its behavior in the steady state.

Figure 7 shows the performance (in terms of total throughput) with different numbers of CPUs for the *libasync-smp* web server. Even though the HTTP server has no particularly processor-intensive operations, we can still observe noticeable speedup on a multiprocessor system: the server's throughput is 1.28 times greater on two CPUs than it is on one and 1.5 times greater on four CPUs.

To provide an upper bound for the multiprocessor speedup we can expect from the *libasync-smp*-based web server we contrast its performance with N inde-

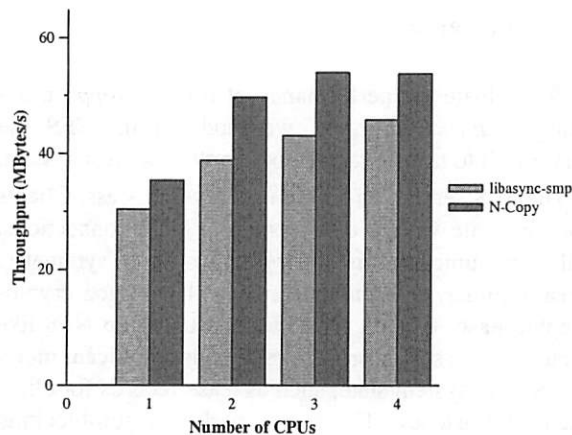


Figure 7: The performance of the *libasync-smp* web server serving a cached workload and running on different number of CPUs relative to the performance on one CPU (light bars). The performance of N copies of a *libasync* web server is also shown relative to the performance of the *libasync* server's performance on one CPU (dark bars)

pendent copies of a single process version of the web server (where N is the number of CPUs provided to the *libasync-smp*-based server). This single process version is based on an unmodified version of *libasync* and thus does not suffer the overhead associated with the *libasync-smp* library (callback queue locking, etc). Each copy of the N-copy server listens for client connections on a different TCP port number.

The speedup obtained by the *libasync-smp* server is well below the speedup obtained by N copies of the *libasync* server. Even on a single CPU, the *libasync* based server achieved higher throughput than the *libasync-smp* server. The throughput of the *libasync* server was 35.4 MB/s while the *libasync-smp* server's throughput was 30.4 MB/s.

Profiling the single CPU case explains the base penalty that *libasync-smp* incurs. While running the *libasync-smp* web server under load, roughly 35% of the CPU time is spent in user-level including *libasync-smp* and the web server. Of that time, at least 37% is spent performing tasks needed only by *libasync-smp*. Atomic reference counting uses 26% of user-level CPU time, and task accounting such as enqueueing and dequeuing tasks takes another 11%. The overall CPU time used for atomic reference counting and task management is 13%, which explains the *libasync-smp* web server's decreased single CPU performance.

The reduced performance of the *libasync-smp* server is partly due to the fact that many of the *libasync-smp* server's operations must be serialized, such as accepting

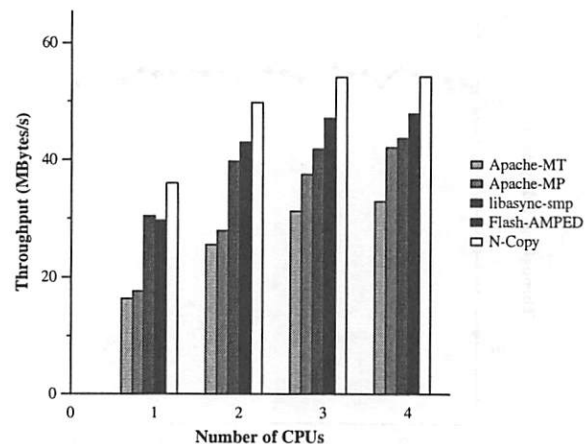


Figure 8: The performance of several web servers on multiprocessor hardware. Shown are the throughput of the *libasync-smp* based server (light bars), Apache 2.0.36 (dark bars), and Flash (black bars) on 1,2,3 and 4 processors.

connections and checking caches. In the N-copy case, all of these operations run in parallel. In addition, locking overhead penalizes the *libasync-smp* server: some data is necessarily shared across threads and must be protected by expensive atomic operations although the server has been written in such a way as to minimize such sharing.

Because the N-copy server can perform all of these operations in parallel and, in addition, extract additional parallelism from the operating system which locks some structures on a per-process basis, the performance of the N-copy server represents a true upper bound for any architecture which operates in a single address space.

To provide a more realistic performance goal than the N-copy server, we compared the *libasync-smp* server with two commonly used HTTP servers. Figure 8 shows the performance of Apache 2.0.36 (in both multithreaded and multiprocess mode) and Flash v0.1.990914 on different numbers of processors. Apache in multiprocess mode was configured to run with 32 servers. Apache-MT is a multithreaded version of the Apache server. It creates a single heavyweight process and 32 kernel threads within that process by calling `clone`. The number of processes and threads used by the Apache servers were chosen to maximize throughput for the benchmarks presented here. Flash is an event-driven server; when run on multiprocessors it forks to create N independent copies, where N is the number of available CPUs

The performance of the *libasync-smp* HTTP server is comparable to the performance of these servers: the *libasync-smp* server shows better absolute performance than both versions of the Apache server and slightly lower performance than N-copies of the Flash server.

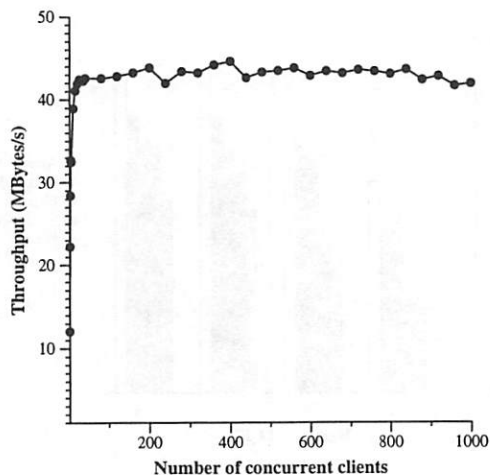


Figure 9: The performance of the web server on a cached workload as the number of concurrent clients is varied.

These servers show better speedup than the *libasync-smp* server: Flash achieves 1.68 speedup on four CPUs while the *libasync-smp* server is 1.5 times faster on four CPUs. Because Flash runs four heavyweight processes, it is able to take advantage of many of the benefits of the N-copy approach: as a result its speedup and absolute performance are greater than that of the *libasync-smp* server. Although this approach is workable for a web server, in applications that must coordinate shared state such replication would be impossible.

Like the *libasync-smp* server, Flash and multiprocess Apache do not show the same performance achieved by the N-copy server. Although these servers fully parallelize access to their caches and do not perform locking internally, they do exhibit some shared state. For instance, the servers must serialize access to the `accept()` system call since all requests arrive on a single TCP port.

The main reason to parallelize a web server is to increase its performance under heavy load. A key part of the ability to handle heavy load is stability: non-decreasing performance as the load increases past the server's point of peak performance. To explore whether servers based on *libasync-smp* can provide stable performance, we measured the web server's throughput with varying numbers of simultaneous clients. Each client selects a file according to the SPECweb99 distribution; the files all fit in the server's cache. The server uses all four CPUs. Figure 9 shows the results. The event-driven HTTP server offers consistent performance over a wide variety of loads.

5.2 SFS server

To evaluate the performance of *libasync-smp* on existing *libasync* programs, we modified the SFS file server [17] to take advantage of a multiprocessor system.

The SFS server is a single user-level process. Clients communicate with it over persistent TCP connections. All communication is encrypted using a symmetric stream cipher, and authenticated with a keyed cryptographic hash. Clients send requests using an NFS-like protocol. The server process maintains significant mutable per-file-system state, such as lease records for client cache consistency. The server performs non-blocking disk I/O by sending NFS requests to the local kernel NFS server. Because of the encryption, the SFS server is compute-bound under some heavy workloads and therefore we expect that by using *libasync-smp* we can extract significant multiprocessor speedup.

5.2.1 Parallelizing the SFS server

We used the `pct[5]` statistical profiler to locate performance bottlenecks in the original SFS file server code. Encryption appeared to be an obvious target, using 75% of CPU time. We modified the server so that encryption operations for different clients executed in parallel and independently of the rest of the code. The resulting parallel SFS server spent about 65% of its time in encryption. The reduction from 75% is due to the time spent coordinating access to shared mutable data structures inside *libasync-smp*, as well as to additional memory-copy operations that allow for parallel execution of encryption.

The modifications to the SFS server are concentrated in the code that encrypts, decrypts, and authenticates data sent to and received from the clients. We split the main send callback-function into three smaller callbacks. The first and last remain synchronized with the rest of the server code (i.e. have the default color), and copy data to be transmitted into and out of a per-client buffer. The second callback encrypts the data in the client buffer, and runs in parallel with other callbacks (i.e., has a different color for each client). This involved modifying about 40 lines of code in a single callback, largely having to do with variable name changes and data copying.

Parallelization of the SFS server's receive code was slightly more complex because more code interacts with it. About 50 lines of code from four different callbacks were modified, splitting each callback into two. The first of these two callbacks received and decrypted data in parallel with other callbacks (i.e., with a different color for every client), and used `cpucb()` to execute the second callback. The second callback remained synchronized with the rest of the server code (i.e., had the de-

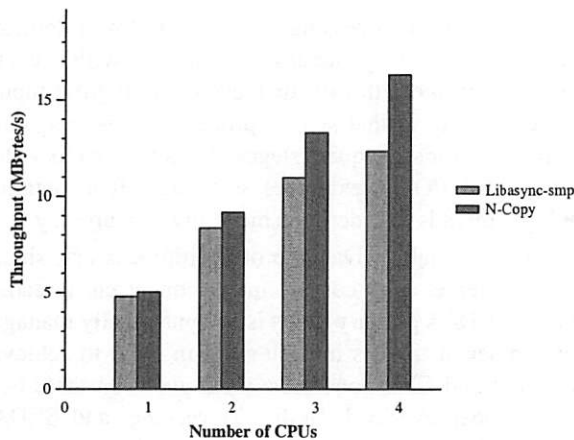


Figure 10: Performance of the SFS file server using different numbers of CPUs, relative to the performance on one CPU. The light bars indicate the performance of the server using *libasync-smp*; dark bars indicate the performance of n separate copies of the original server. Each bar represents the average of three runs; the variation from run to run was not significant.

fault color), and performed the actual processing of the decrypted data.

5.2.2 Performance improvements

We measured the total throughput of the file server to all clients, in bits per second, when multiple clients read a 200 MByte file whose contents remained in the server's disk buffer cache. We repeated this experiment for different numbers of processors. This test reflects how SFS is used in practice: an SFS client machine sends all of its requests over a single TCP connection to the server.

The bars labeled “libasync-smp” in Figure 10 show the performance of the parallelized SFS server on the throughput test. On a single CPU, the parallelized server achieves 96 percent of the throughput of the original uniprocessor server. The parallelized server is 1.66, 2.20, and 2.5 times as fast as the original uniprocessor server on two, three and four CPUs, respectively.

Because only 65% of the cycles (just encryption) have been parallelized, the remaining 35% creates a bottleneck. In particular, when the remaining 35% of the code runs continuously on one processor, we can achieve a maximum utilization of $\frac{1}{0.35} = 2.85$ processors. This number is close to the maximum speedup (2.5) of the parallelized server. Further parallelization of the SFS server code would allow it to incrementally take advantage of more processors.

To explore the performance limits imposed by the

hardware and operating system, we also measured the total performance of multiple independent copies of the original *libasync* SFS server code, as many separate processes as CPUs. In practice, such a configuration would not work unless each server were serving a distinct file system. An SFS server maintains mutable per-file-system state, such as attribute leases, that would require shared memory and synchronization among the server processes. This test thus gives an upper bound on the performance that SFS with *libasync-smp* could achieve.

The results of this test are labeled “N-copy” in Figure 10. The SFS server with *libasync-smp* roughly follows the aggregate performance of multiple independent server copies. The performance difference between the *libasync-smp*-based SFS server and the N-copy server is due to the penalty incurred due to shared state maintained by the server, such as file lease data and user ID mapping tables.

Despite comparatively modest changes to the SFS server to expose parallelism, the server's parallel performance was close to the maximum speedup offered by the underlying operating system (as measured by the speedup obtained by multiple copies of the server).

5.3 Library Optimizations

Table 2 shows how much the use of per-thread work queues improves performance. The numbers in the table indicate how fast a synthetic benchmark executes tasks. The benchmark program creates 16 callbacks with unique colors. Each callback performs a small amount of computation, and then registers a child callback of the same color. The benchmark intentionally assigns colors so that all but one of the task queues are populated, in order to explore the effects of work stealing. The benchmark was run with four CPUs.

The first line shows the task rate with a single task queue shared among all the worker threads. The entry shows the task completion rate when using per-thread task queues. The increase in task completion rate is dramatically higher due to better cache locality, and because there is no contention for the task-queue locks. The third line shows the task completion rate when per-thread task free-lists are used in addition to per-thread queues. The fourth configuration adds work stealing between worker threads. Without work stealing, tasks were never run on one of the four CPUs. Work stealing allows the worker thread on that CPU to find work, at the expense of increased contention for the other threads' task queues.

Library Configuration	Tasks/sec
Base	61420
+ Per-thread Queues	240618
+ Per-thread Task Object Freelists	293997
+ Work Stealing	384765

Table 2: A synthetic benchmark shows improved task processing rates as thread affinity optimizations are added.

6 Related Work

There is a large body of work exploring the relative merits of thread-based I/O concurrency and the event-driven architecture [18, 11, 12, 15, 1]. This paper does not attempt to argue that either is superior. Instead, we present a technique which improves the performance of the event-driven model on multiprocessors. The work described below also considers performance of event-driven software.

Pai et al. characterized approaches to achieving concurrency in network servers in [19]. They evaluate a number of architectures: multi-process, multi-threaded, single-process event-driven, and asymmetric multi-process event-driven (AMPED). In this taxonomy, *libasync-smp* could be characterized as symmetric multi-threaded event-driven; its main difference from AMPED is that its goal is to increase CPU concurrency rather than I/O concurrency.

Like *libasync-smp*, the AMPED architecture introduces limited concurrency into an event driven system. Under the AMPED architecture, a small number of helper processes are used to handle file I/O to overcome the lack of non-blocking support for file I/O in most operating systems. In contrast, *libasync-smp* uses additional execution contexts to execute callbacks in parallel. *libasync-smp* achieves greater CPU concurrency on multiprocessors when compared to the AMPED architecture but places greater demands on the programmer to control concurrency. Like the AMPED-based Flash web server, *libasync-smp* must also cope with the issue of non-blocking file I/O: *libasync-smp* uses an NFS-loopback server to access files asynchronously. This allows *libasync-smp* to use non-blocking local RPC requests rather than blocking system calls.

The Apache web server serves concurrent requests with a pool of independent processes, one per active request [3]. This approach provides both I/O and CPU concurrency. Apache processes cannot easily share mutable state such as a page cache.

The staged, event-driven architecture (SEDA) is a structuring technique for high-performance servers [24].

It divides request processing into a series of well-defined stages, connected by queues of requests. Within each stage, one or more threads dequeue requests from input queue(s), perform that stage's processing, and enqueue the requests for subsequent stages. A thread can block (to wait for disk I/O, for example), so a stage often contains multiple threads in order to achieve I/O concurrency.

SEDA can take advantage of multiprocessors, since a SEDA server may contain many concurrent threads. One of SEDA's primary goals is to dynamically manage the number of threads in each stage in order to achieve good I/O and CPU concurrency but avoid unstable behavior under overload. Both *libasync-smp* and SEDA use a mixture of events and concurrent threads; from a programmer's perspective, SEDA exposes more thread-based concurrency which the programmer may need to synchronize, while *libasync-smp* tries to preserve the serial callback execution model.

Cohort scheduling organizes threaded computation into stages in order to increase performance by increasing cache locality, reducing TLB pressure, and reducing branch mispredicts [14]. The staged computation model used by cohort scheduling is more general than the colored callback model presented here. However, the partitioned stage scheduling policy is somewhat analogous to coloring callbacks for parallel execution (the key corresponds to a callback color). Like SEDA, cohort scheduling exposes more thread-based concurrency to the programmer. Cohort scheduling can also take advantage of multiprocessor hardware.

7 Conclusion

This paper describes a library that allows event-driven programs to take advantage of multiprocessors with a minimum of programming effort. When high loads make multiple events available for processing, the library can execute event handler callbacks on multiple CPUs. To control the concurrency between events, the programmer can specify a *color* for each event: events with the same color (the default case) are handled serially; events with different colors can be handled in parallel. The programmer can incrementally expose parallelism in existing event-driven applications by assigning different colors to computationally-intensive events that don't share mutable state.

Experience with *libasync-smp* demonstrates that applications can achieve multi-processor speedup with little programming effort. Parallelizing the cryptography in the SFS file server required about 90 lines of changed code in two modules, out of a total of about 12,000 lines. Multiple clients were able to read large cached files from

the *libasynch-smp* SFS server running on a 4-CPU machine 2.5 times as fast as from an unmodified uniprocessor SFS server on one CPU. Applications without computationally intensive tasks also benefit: an event-driven Web server achieves 1.5 speedup on four CPUs with multiple clients reading small cached files relative to its performance on one CPU.

Acknowledgments

We are grateful to the many people who aided this work. The NMS group at LCS provided (on short notice) the SMP server used to test an early version of this software. The anonymous reviewers and our shepherd, Edouard Bugnion, provided helpful feedback.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8927.

References

- [1] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKEY, W., AND DOUCEUR, J. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proc. Usenix Technical Conference* (June 2002).
- [2] ANDERSON, T., BERSHAD, B., LAZOSWKA, E., AND LEVY, H. Scheduler activations: Effective kernel support for the user-level management of threads. *ACM Transactions on Computer Systems* 10, 1 (Feb. 1992), 53–79.
- [3] The Apache web server. <http://www.apache.org/>, 2002.
- [4] BIRRELL, A. An introduction to programming with threads. Tech. Rep. 35, Digital Systems Research Center, Jan. 1989.
- [5] BLAKE, C., AND BAUER, S. Simple and general statistical profiling with PCT. In *Proc. Usenix Technical Conference* (June 2002).
- [6] BLASGEN, M., GRAY, J., MITOMA, M., AND PRICE, T. The convoy phenomenon. *Operating Systems Review* 13, 2 (Apr. 1979), 20–25.
- [7] BLOOM, J., AND DUNLAP, K. Experiences implementing BIND, a distributed name server for the DARPA Internet. In *Proc. Summer Usenix Conference* (1986), pp. 172–181.
- [8] BOVET, D., AND CESATI, M. *Understanding the Linux Kernel*. O'Reilly, 2001.
- [9] BURTON, F., AND SLEEP, M. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture* (Portsmouth, New Hampshire, Oct. 1981).
- [10] DABEK, F., KAASHOEK, F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)* (Banff, Canada, Oct. 2001), pp. 202–215.
- [11] DRAVES, R., BERSHAD, B., RASHID, R., AND DEAN, R. Using continuations to implement thread management and communication in operating systems. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 1991), pp. 122–136.
- [12] FORD, B., HIBLER, M., LEPREAU, J., MCGRATH, R., AND TULLMANN, P. Interface and execution models in the Fluke kernel. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Feb. 1999), pp. 101–115.
- [13] FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. Fast and secure distributed read-only file system. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2000), pp. 181–196.
- [14] LARUS, J., AND PARKES, M. Using cohort scheduling to enhance server performance. In *Proc. Usenix Technical Conference* (June 2002).
- [15] LAUER, H., AND NEEDHAM, R. On the duality of operating system structures. In *Proc. Second International Symposium on Operating Systems, IRIA* (Oct. 1978). Reprinted in *Operating Systems Review*, Vol. 12, Number 2, April 1979.
- [16] MAZIÈRES, D. A toolkit for user-level file systems. In *Proc. Usenix Technical Conference* (June 2001), pp. 261–274.
- [17] MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from file system security. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)* (Kiawah Island, South Carolina, Dec. 1999).
- [18] OUSTERHOUT, J. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX technical conference, 1996.
- [19] PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An efficient and portable web server. In *Proc. Usenix Technical Conference* (June 1999).
- [20] SPECweb99 design white paper. <http://www.specbench.org/osg/web99/docs/whitepaper.html>, 2002.
- [21] STEELE, G., AND SUSSMAN, G. Lambda: The ultimate imperative. Tech. Rep. AI Lab Memo AIM-353, MIT AI Lab, Mar. 1976.
- [22] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference* (San Diego, Aug. 2001).
- [23] THOMPSON, S. *Haskell, The Craft of Functional Programming*. Addison Wesley, 1996.

- [24] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2001), pp. 230–243.

Seneca: remote mirroring done write

Minwen Ji, Alistair Veitch, John Wilkes

HP Laboratories, Palo Alto, CA

{minwen.ji, alistair.veitch, john.wilkes}@hp.com

Remote mirroring is widely deployed, but its complexities are surprisingly poorly understood. This means that data is less well protected than it ought to be, and possible optimizations are infrequently taken advantage of. To address these difficulties, we (1) demystify the design space by presenting a taxonomy of the approaches in use, (2) describe Seneca – a new asynchronous remote-mirroring protocol that supports write coalescing, asynchronous propagation, and in-order delivery, and (3) report on a performance and correctness validation of Seneca. We are confident that the result is a robust remote-mirroring protocol that provides good performance and predictable behavior in the face of a wide range of failure types, such as rolling disasters.

1 Introduction

Data mirroring is a classic technique for tolerating failures: by keeping two or more copies of important information, access can continue if one of them is lost or becomes unreachable. It is used inside disk arrays (where it is called RAID1), between disks or disk arrays, and across multiple sites, where it is called *remote mirroring*.

Remote mirroring is widely deployed whenever the cost of losing data matters. And it does matter: protection for information assets is often more important than for physical ones – at least the latter can be replaced after a loss. Gartner estimates that “Two out of five enterprises that experience a [site] disaster ... go out of business within five years” [Witty2001]. Even lack of access to data is expensive: 25% of respondents to one survey estimated that outages cost them more than \$250k/hour, with 4% estimating more than \$5M/hour [EagleRock2001]. Remote mirroring can protect against both data loss and inaccessibility.

The design choices for remote mirroring are complicated by competing goals: keeping the copies as closely synchronized as possible, delaying foreground writes as little as possible, maintaining accessibility in the face of as many failure types as possible, and using as little expensive inter-site network bandwidth as possible.¹

The basic trade-off is between better performance with lower cost against greater potential data loss, especially for recently-written data. Simple solutions, such as synchronously updating all copies, provide high resilience to data loss but have poor write performance and incur high network costs in remote-mirroring systems.

Contributions of this paper

Remote mirroring has been in use for quite a while, so it is usually thought to be well understood. Despite this, when we prepared a survey of the approaches used in practice, we found a wide variation in assumptions, techniques used, and the degree to which recovery is achievable. Additionally, many of the design choices are quite intricate and subtle, as we discovered when we ran our taxonomy past practitioners in the field. Our first contribution, then, is a taxonomy of the design choices for remote mirroring.

Our second contribution is the design of a robust remote mirroring protocol that provides resilience to many kinds and sequences of failures, low network bandwidth demands, and low (and tunable) data loss. We also look at its correctness, using an I/O automata-based simulation.

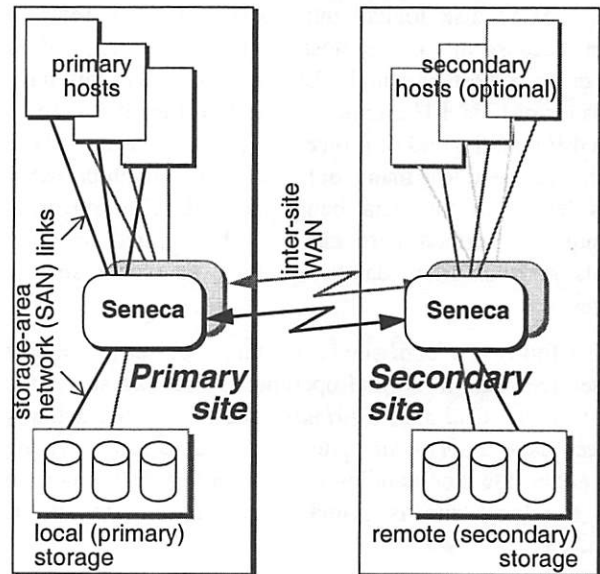


Figure 1: canonical remote mirroring system. The remote-mirroring function (labelled “Seneca”) can be implemented in hosts, disk arrays, or in SAN appliances (as shown).

¹ For example, in South Carolina, the rental price of an 155Mb/s OC3 line in 2002 was about \$460k/year [SC2002a]. This is equivalent to the capital depreciation cost of about 12TB of mirrored, enterprise-class (HP EVA) storage at late 2002 prices. Slower line costs are scaled accordingly: constant bit-rate ATM lines inside South Carolina are priced at an average of \$170 per Mb/s per month [SC2002b]; variable-rate a bit less.

In California, SBC Pacific Bell [SBC2002] offered ATM service for: DS1: 1.544Mbps = \$750 per month plus one-time installation of \$1200 DS3: 40Mbps = \$5000 per month plus one-time installation of \$3000 OC-3c: 148Mbps = \$7000 per month plus one-time installation of \$3000.

This 6:1 cost ratio is a result of the rapidly-changing telecom market, and is considered “not unusual” by our IT manager.

The final contribution is an evaluation of the protocol's performance, looking at how much it reduces network traffic using traces from real workloads.

2 A taxonomy for remote mirroring

The usual assessment criteria for remote mirroring designs include data accessibility (availability), resistance to loss or corruption of data (reliability), performance degradation in normal use, and the cost of operation, primarily in terms of inter-site network traffic. Accommodating these conflicting goals exposes many design and configuration options for remote-mirroring. The basic axes are as follows:

- the fault-coverage model;
- how closely synchronized the copies are;
- how updates are propagated;
- when updates are acknowledged;
- where the data duplication is performed.

All of these affect write performance and the amount of recently-written data that can be lost; some of them affect the amount of network traffic needed, too.

We use SCSI disk logical units (LUs) as the entities to mirror, because this is the most common practice. A SCSI disk drive exports a single LU; a disk array can have thousands of LUs. LU granularity allows different data to be given different degrees of protection: for example, a user file system may need less than a database index, which probably needs less than the data being indexed. The mirroring techniques described here can also be applied to other objects such as files, database tables, or object storage devices.

In what follows, a *local* site is one that is closer to the host and services data in normal operations; a *remote* site is the mirror of the local site; a *primary* site is one that actually services data; a *secondary* site is one that is not servicing data, either down or standing by as a backup. In the normal case, the local site is primary and the remote site is secondary.

2.1 Fault model

The fault model we used when designing the Seneca protocol is representative of those used in most remote mirroring designs – albeit more comprehensive than many of them. A remote mirroring system should tolerate failures of:

- host computers (hardware and software);
- links, switches, and hubs at each site (these comprise the local Storage Area Network, or SAN);
- wide- or metropolitan-area links between sites (WAN, MAN);
- any dedicated hardware used to implement remote mirroring;

- storage devices (but we ignore failures that are masked completely within a disk array);
- an entire site.

We assume fail-silent failures, and that recovery or repair of failed components is possible. A full-scale site disaster may take days to months to recover from, while a site power outage may be corrected in minutes, and a broken long-distance link may recover in a few seconds.

The traditional approach is to mirror storage across two sites, but more sites are possible, and may even be mandated soon for the financial services industry. We concentrate on the 2-site case here to simplify the exposition. The physical separation between sites is governed by the kinds of site failures that are to be tolerated. For example, a fire may take out a single building, a power outage all the buildings on a single campus, an earthquake or flood all the buildings within a metropolitan area.

Both repeated and multiple concurrent failures are expected. One failure can cause multiple components to fail (e.g., if it is in a shared component such as a power supply or air conditioning unit), or it can trigger a cascade of failures by increasing the stress on the rest of the system – including its operators.

We exclude certain scenarios: multiple concurrent site failures; pervasive software design or implementation faults, such as ones that fail to maintain duplicated copies correctly; and mis-installations. These may lead to a *disaster*, which we define as unacceptable data or availability loss, as may the occurrence of “too many” failures that occur before recovery actions from a previous failure can be completed. In all cases, data loss is less tolerable than lack of data availability.

2.2 Bounded divergence

Minimizing the risk of losing recently-written data means updating remote copies as rapidly as possible. This is easily achievable when the two copies are physically close (e.g., within the 10km single-link Fibre Channel distance limit), but becomes problematic if they are further apart, when the time to propagate an update across a long-distance link can be prohibitive. For example, the best-case speed-of-light round trip time across the continental USA is about 27ms, which is larger than a typical disk access, and huge compared to the access time to a disk array's cache. As a result, there are strong incentives to overlap the propagation with subsequent I/Os. The drawback is reduced reliability: writes that haven't been propagated to the remote site will be lost if the primary goes down.

Delayed propagation of updates can also reduce the worst-case long-distance network traffic needed because write-buffering allows bursty write traffic to be spread out more evenly over time [Ruemmler1993].

This leads us to the first part of our solution taxonomy: the amount by which the copies are allowed to diverge while the links between sites are up. The basic choices are “none” or “some”. If the remote site becomes unreachable in the “some” case, or updates aren’t propagating fast enough, there are two choices as to what to do at the primary: stall (or abort) writes until the secondary copy catches up, or switch to a mode with a looser divergence bound.

No divergence: all updates (writes) are propagated immediately to the remote site. This implies one of the *lock-step* protocols described below. Writes stall or fail if the remote copy is unreachable. *Analysis:* the safest thing to do – and the slowest. This is part of what most people mean when they talk about “synchronous mode”; it is the only mode in which non-catastrophic multiple failures will essentially never lose data: in all other modes (usually called *asynchronous remote mirroring*), data that hasn’t propagated to other sites could be lost.

Operation- and/or byte-count divergence: the maximum divergence between the copies is deliberately limited to a small, fixed number of I/Os (e.g., one outstanding remote I/O per LU in EMC SRDF’s “Semi-Synchronous” mode [EMC-SRDF]), or a bounded quantity of data. *Analysis:* this allows the transfer time to the remote site to be overlapped with a local I/O, which offers marginally better performance than the above. If the operation count or byte count is small, then the amount of data subject to loss is well bounded, so recovery may be simplified.

Resource-bounded divergence: the amount of divergence is bounded by some resource’s size (e.g., the size of a log file or disk, or the size of an array’s NVRAM cache, as in the HP XP1024 disk array [HP-XP1024, HP-XP-CA]). *Analysis:* even moderate amounts of divergence allows both good local performance and good inter-site link bandwidth needs. In practice, the limits are rarely met: disk array caches are often measured in gigabytes, and log files can easily be made much larger. Such systems still have to operate correctly when resource bounds are met, of course, and most do so by dropping into the unbounded divergence state discussed next. The order in which changes are propagated to the remote site matters, too – this aspect just captures how much divergence is allowed.

Unbounded divergence: there is no bound on the amount of divergence allowed. Faced with an unreachable site, and/or lack of log space for updates, this is all that can be done if local writes are to continue. It is common to keep track of which data has been updated (e.g., using a per-block, -track, or -cylinder data structure such as a bitmap) to reduce the amount of data that has to be sent over the link when the sites reconnect. *Analysis:* this allows the best performance, and the maximum amount of data-accessibility in the case of

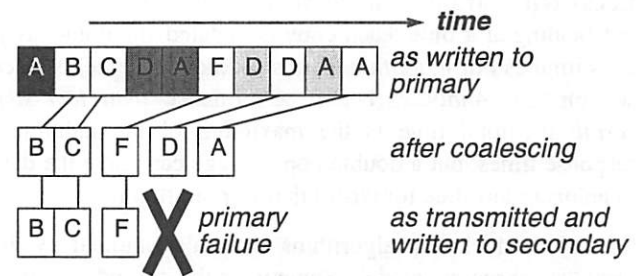


Figure 2: rolling disaster from overwrites. The top row shows a sequence of data blocks being updated; shading indicates generations of overwritten blocks. The second shows the data blocks left after coalescing. The third shows the state of updates after half the data has been transmitted; no updates to blocks A or D have been propagated, so the result is not consistent.

single failures: access is not denied unless the data is really unreachable. Once this state is entered, the only way to repair a remote copy is to propagate *all* the changes.

2.3 Single-LU propagation order

Write buffering enables *write coalescing* (or *overwrite absorption*): the overwriting of older writes in the queue of waiting data. This is most often used at the primary to reduce network traffic, but it could also be applied at the secondary to reduce the amount of work needed for an update. Even small amounts of memory for write coalescing can be quite effective [Ruemmler1993].

The primary effect of write coalescing on fault tolerance is to alter the order in which updates are applied at the secondary site. The goal is to ensure that the secondary is always at some *consistent state*—one that could be reached by a prefix of the sequence of writes applied at the primary. A failure part-way through applying a reordered set of updates can leave the remote site in a non-consistent state when it needs to take over from the primary. If the primary site is lost at this point, it may not be possible to recover the application that uses the data. (See Figure 2).

To simplify the discussion, we begin by considering the ordering options that apply to a single LU, and defer multiple-LU cases to section 2.4. The design choices are ordered by the amount of reordering, parallelism, and coalescing they permit.

Lock-step serial: at most one host write can be outstanding at a time; each copy is updated in a known order. This implies *full write-though* host acknowledgements – see section 2.5. *Analysis:* this provides the most careful form of dissemination for updates, and allows recovery from double outages, where the write order is otherwise not knowable. It is only appropriate when reliability is much more important than performance. The total time to perform the write is the sum of the response times of the writes to each copy.

Lock-step parallel: at most one host write can be outstanding at a time; each copy is updated simultaneously. This implies *full write-though* host acknowledgements – see section 2.5. *Analysis:* better performance than *lock-step serial*; the total time is the maximum of the individual response times, but a double-copy outage can leave the data in an uncertain state for writes that were in flight.

One of the lock-step algorithms is usually implied by the term “synchronous mode”. Sometimes the secondary site is updated first (in case connection is lost, and the write should then be aborted); sometimes the primary. Combinations may also be useful, such as writing to the primary copy first, and then updating multiple secondary copies in parallel.

In-order asynchronous: here, the updates are propagated to, and applied at, the remote sites in the same order that they occurred at the primary. *Analysis:* this does nothing to reduce the amount of WAN traffic, although it can smooth out the traffic from write bursts if the divergence bound (i.e., delay pipeline) is large enough.

Dependency-preserving asynchronous: writes are provided with explicit dependency information, and propagation preserves these, but is otherwise free to reorder the updates. *Analysis:* this allows better performance by exploiting asynchrony where it would preserve the update semantics. We do not believe that this has been implemented for remote mirroring systems, probably because of the difficulty of providing the necessary dependency data, even though host-based variants have been shown to work well [Kondoff1988].

Write-coalescing batches with atomic update: this scheme explicitly delays sending a *batch* of updates to the remote site, in the hope that write coalescing will occur, and only one copy need be propagated. To avoid inconsistent states at the secondary, writes can be coalesced only within a batch, and batches must be committed atomically at the remote site: that is, all or none of the updates in the batch must be applied. *Analysis:* if overwrites are common, this can greatly reduce the amount of WAN traffic, at the expense of losing more updates if the primary site fails.

The size of the batch can be selected in several ways, such as the elapsed time, the number of updates, or the amount of data written, or bytes to transfer. It would be possible to select the batch size that achieves a particular target data-loss likelihood, taking the WAN link reliability into account (cf. AFRAID [Savage1996]).

Batches can be implemented by logs, or (at a coarser grain) with an array-based *LU snapshot* mechanism (e.g., EMC TimeFinder [EMC-TimeFinder], HP business copy XP [HP-XP-BC]). A snapshot is a virtual copy of an LU, and updates it, using copy-on-write, whenever the original LU is modified. Since the update traffic needed to implement a

snapshot is primarily a function of the original LU's update rate and the snapshot's lifetime, not the total amount of data in the original LU, keeping a snapshot up to date is reasonably quick, and it is often possible to synchronize one in only a few minutes. (Some snapshot implementations take space proportional to the differences, so creating these is quick, too; some, like EMC TimeFinder copies, are complete mirror copies that can be incrementally updated: creating them is slow, but updating them can be done in time proportional to the amount by which they have diverged).

All atomic update schemes require sufficient buffering at the primary and the secondary to store the largest possible batch – which may be as large as the complete data size. Obviously, this isn't always possible (or cost-effective), so most schemes revert to non-atomic propagation when the divergence limit switches to unbounded.

Overwrite-log with atomic update: the overwrite-log scheme removes the hard send-batch boundaries in favor of a log of updates that allows write coalescing, together with the use of *receive batches* which are defined as the updates that occur between the first write of a data block and its overwrite, plus that overwrite. Receive batches represent the smallest unit that must be applied atomically at the secondary site. *Analysis:* this provides better behavior than the batched scheme, which wait until the end of the batch has been declared before they start propagating any data within the batch, perhaps delaying transmission longer than is needed, and unnecessarily exposing data to primary site failure. Seneca is the only example we know of.

Send barriers are used to mark the points beyond which write coalescing may not occur. These are needed to prevent the size of receive batches growing out of hand—but they need to be inserted only when necessary (e.g., to mark the end of a receive batch that has commenced transmission). For example, one simple dynamic scheme is to start transmitting pending updates immediately if the WAN is lightly used, or to artificially delay updates if the WAN is busy, to increase the overwrite rate.

Out-of-order asynchronous: the order in which updates are propagated to the remote site is unrelated to the order in which they occurred (e.g., a bitmap of updated tracks is kept, and the updates are propagated in track-number order). This case is implied if the primary is forced into the unbounded divergence case. *Analysis:* good performance, but a rolling disaster (see section 2.4) can occur if the primary site is lost before the secondary site has been fully updated.

Complete copy replacement: the old copy is simply overwritten in its entirety. *Analysis:* this may make sense if the number of changes to be applied is large (such as when a mirror copy was detached, most of the primary was updated, and now the secondary is being brought back into synchrony – a process sometimes called *resilvering*); if a partial failure

during the resynchronization is immaterial so the copy can simply be restarted to recover (e.g., if the copy is only being used to make a backup from); or if the expense of maintaining state about the differences is too high.

Explicitly delayed: instead of propagating changes as quickly as possible to a remote site, they can be explicitly delayed by at least a certain period of time, or until some event has occurred, such as a sanity check. For example: “keep 3 copies: one is the master; one is a remote copy that is as up to date as possible; and one is always as close to 12 hours behind the master as possible.” *Analysis:* this technique can be used to allow time to run sanity checks (e.g., virus scans) over the data before installing it at the secondary.

Explicit delays might be used to replace backups, which are necessary today to guard against data corruption and operator error, both failures that mirroring does nothing about. (Indeed, most mirroring schemes try hard to propagate errors at the speed of light!)

2.4 Multi-LU propagation order

If an application updates two LUs, on the same or different arrays, and there are consistency requirements between those updates, then additional steps have to be taken to handle these needs. It is common to use a *consistency group* to specify the LUs involved. The important thing is the degree of guarantee offered by the consistency group:

No inter-LU consistency. *Analysis:* the name *rolling disaster* says it all—and comes from a sequence of storage device failures at different times (e.g., as the result of a fire or flood in a data center), or from the case where some of the inter-site links fail but others do not. In either case, the secondary site can end up in an inconsistent state.

Single-array multi-LU sequencing: maintaining the relative write order for updates across the LUs, e.g., by using a single propagation queue. *Analysis:* this works well for data whose primary copy is stored within a single array, but offers no guarantees at all otherwise. All of the single-LU propagation options discussed above apply here, too.

Multi-array propagation-cessation: since inconsistencies only occur if writes are propagated out of order, it is sufficient to stop sending updates to the secondary sites as soon as *any* write cannot be delivered. This can be achieved by having the application (or host OS) stop writing, or the disk arrays stop propagating writes, as soon as a propagation failure occurs. *Analysis:* there may be a small window of vulnerability if there are multiple outstanding independent updates while propagation-failure is detected, but this window can be bounded to roughly the duration of a long-distance timeout, or by using lock-step propagation (perhaps in conjunction with last-update rollback at the remote sites.

Implementations of this approach typically require software support in the host systems (e.g., [EMC-CG2002]).

Multi-array barrier-atomic propagation: multiple disk arrays collaborate to generate *synchronization barriers*, which act to prevent updates that occur after such a barrier being applied before *all* updates that preceded the barrier have been propagated. This can be achieved via a two-phase commit protocol across the primary copy arrays (plus the hosts, if there are more than one), followed by a 2-phase commit across the secondary copies before updates are applied. *Analysis:* this is the most desirable state of affairs, but as the description suggests, it probably cannot be achieved without higher overheads and complexity than the preceding cases.

2.5 Returning acknowledgements to the host

The descriptions above considered the degree of synchronization from the point of view of the copies. Another perspective is the viewpoint of the host: just how soon is it told “we have it” on a write? Sooner means greater best-case performance, because it allows the maximum amount of concurrent I/O activity; later leaves fewer opportunities for things to go wrong after the host believes the write has been successfully recorded.

If the host issues multiple writes at a time for a single LU, the SCSI standard allows the disk array to service these in any order it finds appropriate, provided the effect is as if each write was completed in the order indicated by the request-completions sent to the host. If such writes are serviced in a different order at the primary and secondary, certain failures may make this visible.

It is helpful to separate the volatility of data from its location; in what follows, we concentrate on the volatility, remembering that these observations can be applied at both the primary and secondary copies.

Volatile immediate-report: the write is acknowledged as soon as the data is received into any storage system memory, even if this is volatile RAM. *Analysis:* Best possible performance; but data is vulnerable to a single failure, including power loss. SCSI disk drives offer this mode for data written to their cache, but it is only of use if the upper level software knows what it is doing, and it may have to explicitly sequence writes (e.g., by draining the I/O pipeline, or controlling the command queuing).

Non-volatile immediate report: the write is acknowledged as soon as the data is received into at least one non-volatile memory (NVRAM) in the storage system. *Analysis:* This is the mode commonly used by single-controller disk arrays. Data is vulnerable to loss of the single copy.

Redundant non-volatile immediate report: the write is acknowledged as soon as the data is written into failure-

tolerant non-volatile memory in the storage system (e.g., mirrored NVRAM). *Analysis:* This is the mode commonly used by high-quality dual-controller disk arrays. Data is still vulnerable to loss if the memory loses data, if the disk array suffers a complete failure, or if the site goes down.

Single write-through: the write is acknowledged only after the data has been transferred onto at least one of the long-term storage devices. *Analysis:* This is the mode used by asynchronous remote mirroring: the primary copy is the target for the write-through, the secondary, remote copy happens later. Data is vulnerable to loss of the local copy, or a site failure.

Redundant write-through: the write is acknowledged only once the data has been written through to “sufficiently many” long-term storage devices to survive the target number of concurrent covered failures. *Analysis:* redundancy can be provided by updating the local and remote copies, or by updating the primary copy and the write-behind log of pending data that needs to be sent to the secondary, remote, copies. Data is vulnerable to loss of all the copies at a site, or a site failure.

Full write-through: the write is acknowledged only once all copies have been updated. (This mode implies *lock-step* synchronization.) *Analysis:* This is the mode commonly called “synchronous”. Reliability is as good as it gets; performance less so. There are circumstances when it’s still the right thing to do.

2.6 Where data duplication is done

Remote mirroring requires data duplication, which can be performed in four main places, each with advantages and disadvantages. Although the relative importance of these factors changes, common concerns include: the ease of supporting heterogeneous hosts and storage devices; whether additional hardware elements are needed; whether host software needs changing or installing; additional host CPU loads; cost, performance, and scalability of the solution; and ease of supporting LU groups that span disk arrays (simplest at the host, hardest at the arrays).

At the host: typically in a device driver or logical volume manager (LVM). *Analysis:* typically gives the greatest flexibility in terms of WAN link support; allows for file-level replication, not just at the LU level; simplifies the grouping case; but imposes additional CPU load on the hosts, and may be harder to manage if the ratio of hosts or host types to storage devices is large.

In an I/O card at the host: typically by means of a “smart” host bus adapter (HBA). One example is the original COMPAQ VersaStor scheme [Widen2000]. *Analysis:* can off-load I/O processing from the host, and avoids OS-dependence of the host-based scheme. Relies on the SAN to

provide in-band connectivity to inter-site links, e.g., by running IP over Fibre Channel to a gateway.

At the disk array: typically in the array controller firmware. Current implementations tend to require the remote array to be from the same manufacturer (and sometimes the same model), as the primary array. *Analysis:* probably the most commonly deployed scheme today. In the past, fewer native network-link types were supported than with host-based approaches; more recently, gateways and protocol converters have broadened the scope of support.

In the SAN fabric: either as an in-band “SAN appliance”, or as an extension to a SAN fabric switch. An example of the former is the HP CASA product [HP-CASA]. *Analysis:* conceptually elegant, but may be limited by the I/O bandwidth or latency of the in-band hardware. Bounding the traffic to just the LUs in a group (in the sense of section 2.4) can be used to achieve scalability: one box needs only to handle the traffic of a single group, or a small number of them. It’s also important that the appliance doesn’t become a single point of failure: most appliance implementations support failover pairs for this reason. Switch-based implementations are actively being discussed in industry; how failure-tolerance will be achieved in this case is not yet clear.

2.7 Additional features

In this section, we summarize some additional techniques and possible extensions.

Multiple secondary copies. Although we have chosen to focus on the 2-copy case for ease of exposition in this paper, it’s clear that there can be more than one secondary copy, and the different copies can have different techniques applied to them.

Partial copies. The descriptions above are written as if each copy is a full instance of the data. There are circumstances in which this need not be the case: what matters is that the mirroring system can provide the illusion of a full copy, even in the face of failures. Thus, an old copy plus a redo log can be used to provide the illusion of an up-to-date copy and a prior one. So can a current copy and an undo log. Similarly, just the log of recent changes can itself act as a virtual copy if there are other mechanisms for restoring the underlying state: this means that a 2.5-copy system could be constructed with the log held at an intermediate site, separate from the true secondary copy.

Bidirectional mirroring: some systems allow both ends to act as primaries for different LUs.

Active-active mirroring: two or more sites can update the same LUs. This requires both synchronous mirroring and application-level support.

<i>system</i>	<i>divergence</i>	<i>propagation order</i>	<i>where done</i>
Veritas Volume Replicator	none / IO-count-bounded / log-space-bounded / unbounded (bitmap)	lock-step / in-order asynchronous / in-order asynchronous / out of order	host
IBM's Peer-to-Peer Remote Copy (PPRC)	none / unbounded (full copy)	lock-step / out of order	disk array
IBM's Extended Remote Copy (XRC).	space-bounded / unbounded (full copy)	in order asynchronous / out of order	host + disk array
EMC symmetrix SRDF	<i>Synchronous</i> : none / unbounded (per-track bitmap)	lock-step / out of order	disk array
	<i>Semi-synchronous</i> : IO-count-bounded (≤ 1) / unbounded (track "bitmap")	in-order / out-of-order	disk array
	<i>Adaptive Copy</i> : track-count-bounded / unbounded (track "bitmap")	out of order / out-of-order	disk array
NetApps SnapMirror	unbounded (file system structure)	atomic in-order asynchronous batched, with overwrites	file server
HP XP continuous access	<i>synchronous</i> : none / unbounded (bitmap)	lock-step / out of order	disk array
	<i>asynchronous</i> : cache-space-bounded / unbounded (bitmap)	in-order asynchronous / out of order	disk array
HP Continuous Access Storage Appliance (CASA)	<i>synchronous</i> : none / unbounded (bitmap)	lock-step / out of order	duplexed SAN appliance
	<i>asynchronous</i> : disk-queue-space-bounded / unbounded (bitmap)	in order / out of order	duplexed SAN appliance
Seneca	time-bounded batches / log-space-bounded / unbounded (bitmap)	grouped, atomic in-order asynchronous batched, with overwrites / (same) / out of order	duplexed SAN appliance

Table 1: a few sample remote mirroring systems. A “/” between steps indicates the next level of fallback behavior. Additionally, in most cases, synchronous mode can be told to allow or fail (abort) application I/Os if the link goes down; these variants are not shown.

Multiple recovery points: both the batch- and log-based approaches lead themselves to a strategy of preserving prior state, rather than discarding it as soon as an update is applied. This can permit state reconstruction at multiple points in the past. Tape backups can be thought of as one extreme form of this; the S4 project at CMU, which keeps the complete write log, another [Strunk2000].

2.8 Application fail-over

We have concentrated here on the recovery and propagation behavior of the storage system. In real life, the recovery and failover behaviors of the applications are also vital, but it is beyond the scope of this paper to address these aspects, other than to observe that recovering a consistent copy of the stored data is but the first step to application recovery.

2.9 Applying the taxonomy

We found it instructive to test the taxonomy by applying it to a number of extant remote mirroring products. Table 1 provides such a sample. In addition, we expand on a few of

them here to illustrate the remote mirroring design space in a little more detail.

EMC SRDF: EMC claims to have been the first to market with a “storage-based replication software application” in 1993 with SRDF, the Symmetrix Remote Data Facility [EMC2002, EMC-SRDF], which provides inter-array remote mirroring. Up to two remote copies can be maintained from one primary using the “Concurrent SRDF” feature, in synchronous mode.

EMC literature encourages the use of synchronous mode, which is a lock-step, no-divergence protocol. When the performance penalty of synchronous operation is too high, semi-synchronous mode is available, which allows one outstanding I/O per LU to be in flight asynchronously to the secondary site. Subsequent writes are stalled until it returns. Adaptive copy mode provides track-count-bounded asynchrony, with no ordering guarantees; if the count is exceeded, writes are stalled.

Multi-array propagation cessation is supported in synchronous mode, with help from EMC-supplied host software [EMC-CG2002].

If the remote site becomes disconnected, a per-track data structure keeps a note of which tracks have been modified – but not the order in which this is done (i.e., this is effectively a bitmap-like solution), so recovery from any extended link break or site outage is always performed in unordered mode, which is vulnerable to a rolling disaster. To avoid this, EMC suggests the use of the separate TimeFinder product to provide the effect of (large) batches.

IBM XRC: IBM's Extended Remote Copy (XRC) software for their MVS mainframes employs a host-based data mover to drain cached data from disk arrays' NVRAM cache, write it to an on-disk log, and transmit it to a remote site [IBM1997]. Each data mover session groups and orders the I/Os for the volumes under its care in timestamp order. If the disk array cache fills up, application I/Os are stalled, up to some threshold time, after which XRC reverts to the unbounded case, and a full volume copy will be needed.

HP XP arrays: the HP XP disk arrays and the similar Hitachi Data Systems arrays provide both synchronous updates and an in-order, space-bounded asynchronous update mode [HP-XP1024, HP-XP-CA]. The latter mode keeps writes in the mirrored NVRAM cache of the primary array; the ordering is implemented by maintaining a timestamp or sequence number on dirty blocks in the cache, and applying the writes in the same order at the secondary. Multiple LUs in one array can be grouped into a consistency group, which is the unit of I/O ordering. The amount of cache allocated to the write buffer is sized dynamically; when it gets above a high water mark, foreground application writes will be deliberately slowed down in increments, eventually to virtual lock-step. As we will show later, the array's cache is likely to provide adequate buffering in the absence of a link or remote site failure without any slowdown. As a last resort, the array reverts to bitmap-based unbounded divergence.

HP Continuous Access Storage Appliance (CASA): a SAN mirroring appliance [HP-CASA]. In asynchronous mode, it provides in-order delivery until a disk-based queue fills up, and then resorts to an unbounded bitmap mode to track updates during any remote site or link failure.

Veritas Volume Replicator (VVR) [Veritas2002]: a host-based scheme that can support multiple remote copies. It can operate in synchronous mode, or with per-host I/O-count-bounded, multi-LU, in-order asynchrony, using a circular disk-based "storage replication log (SRL)". If the remote site becomes unreachable in synchronous mode, then writes can be refused, or VVR can drop into "soft" asynchronous mode until the link comes back up. If the SRL fills up, writes can be stalled until space is available, refused, or VVR can drop

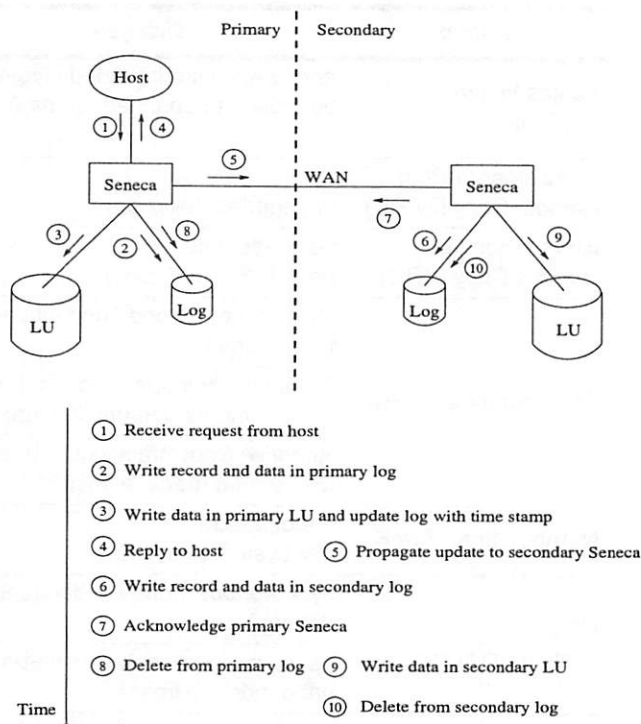


Figure 3: the general steps in the Seneca mirroring protocol.

into an unbounded mode that uses a bitmap to record updated blocks. If a count-based bound for asynchronous I/Os is reached, writes can be refused or stalled until a low water mark is reached. VVR does no overwrite absorption in normal operation, and nor does it perform atomic updates at the secondary. Writes are acknowledged from the secondary copy as soon as they are received in the secondary host (a special case of the volatile immediate report of section 2.5).

Network Appliance SnapMirror: this product uses asynchronous, write-coalescing batched file system updates that are applied atomically at a remote file server [Patterson2002]. The WAFL file system is used to keep track of the blocks that have been updated, including which blocks are still in use and worth propagating. The WAFL file system always operates in a no-overwrite mode, so it readily supports applying a coordinated set of updates atomically.

3 The Seneca protocol¹

For Seneca, we set out to design an asynchronous mirroring system that could exploit a relatively low-speed wide-area network link by coalescing overwrites safely – i.e., in an order-preserving manner. Seneca's goal is to make the data available and keep each copy consistent despite disk array

¹ "Seneca, comparing his degenerate times with those of the heroic Scipio, who bathed in austere simple surroundings, complained: 'But who in these days could bear to bathe in such a fashion? We think ourselves poor and mean if our walls are not resplendent with large and costly mirrors ...'" [de la Croix1975, p225].

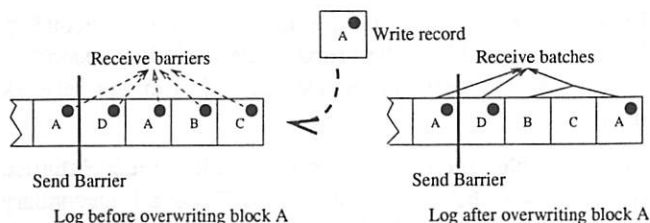


Figure 4: receive batches.

failures, Seneca box failures, network failures, and both temporary and permanent site outages.

To simplify the exposition, we present the Seneca protocol as if it were running between a dedicated pair of remote-mirroring SAN appliances (Figure 1), although the protocol could readily be implemented in the disk arrays, in the hosts, or in host I/O bus adapters.

An overview of the protocol is shown in Figure 3. In normal operation (i.e., while the remote site is reachable), a write at the primary causes a *write record* to be written synchronously in a *primary log*, and the data can then be written to the primary copy (LU). The primary log can be implemented in a disk inside a Seneca box, or, better, in a disk array in the local SAN—preferably one equipped with an NVRAM write buffer, and disjoint from the one holding the primary copy.

Seneca implements the overwrite log with an atomic update protocol. It can use either the single or redundant write-through schemes to report completion back to the host; and it can be integrated into a multi-LU propagation scheme.

3.1 Log barriers

Update records and data blocks in the primary log are propagated to the secondary Seneca in the same order as they were written to the primary LU. This transmission occurs in parallel with continuing operations at the primary; we assume in-order transport across the WAN. The propagated blocks are appended to the log at the secondary Seneca. This *secondary log* is divided into *receive batches*, which are committed to the secondary LU atomically. Receive batches are bounded by inserting *send barriers* and expanded by removing *receive barriers*.

A *send barrier* is one that Seneca deliberately and periodically inserts into the primary log, following the last block that has been written to the primary LU. Overwrites to blocks following the last send barrier are allowed, which saves both network bandwidth and log space. Overwrites to blocks preceding any send barriers, however, are prohibited. Send barriers may bound the divergence between the primary and secondary Senecas in terms of the elapsed time, the number of transactions, the amount of data, or any other metric. Their frequency can be adjusted manually or automatically, as described in section 2.3.

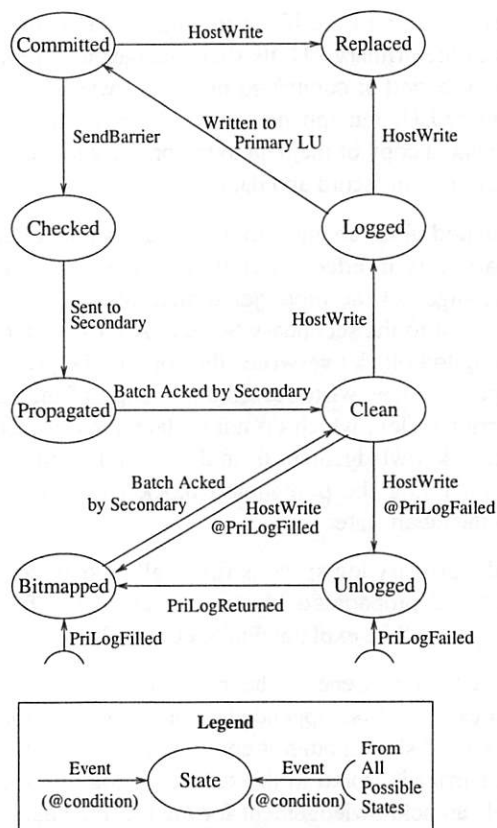


Figure 5: data block states in the primary Seneca.

A *receive barrier* is initially associated with each write; that is, each data block can by default be committed at the secondary Seneca by itself. Receive barriers bound the sets of blocks that have to be applied atomically as a unit. When a block is overwritten before it is propagated, the earlier write record for that block is removed from the log, as are any receive barriers for blocks written between the old copy and the new write (i.e., the end of the log). This merges those blocks into the same receive batch.

For an example, consider Figure 4. The second write to data block A overwrites the earlier version, and removes the receive barriers associated with blocks B and C that were written between the two writes to A. The consequence of the overwrite is that the three blocks A, B and C can no longer be transferred to or committed in the secondary Seneca in the same order as they were written—instead, either all of them need to be committed at the secondary copy, or none.

3.2 Block state transitions

Figure 5 shows the state transitions of a data block and its write record in the primary log.

A data block is in the clean state if it has not been written since the last time it was committed at the secondary Seneca. A write to a clean block normally adds a *write record* and a copy of the newly-written block to the primary log and

changes the state of the block to logged. After the block is written to the primary LU, its state changes to committed. A write to a logged or committed block overwrites the copy in the primary LU, and appends a new write record (with state logged) and a copy of the data to the primary log, effectively replacing the old record and data.

A committed block changes to the checked state after a new send barrier is inserted (after it) into the log. A checked block changes to the propagated state after its record and data are sent to the secondary Seneca. A write to a checked or propagated block overwrites the copy in the primary LU, and appends a new write record and a copy of the new data to the primary log, which do not replace the old record and data. An acknowledgement from the secondary Seneca for a batch containing the propagated block changes the block back to the clean state.

When the primary log space is filled, all logged, committed, checked and propagated blocks change to the bitmapped state, which will be explained in Section 3.3.

At the secondary Seneca, the newly-received records and data blocks are first appended to a *secondary log* there, typically on disk, although it could be in NVRAM. When a receive barrier is stored in the secondary log, the secondary (1) sends an acknowledgement for the receive batch back to the primary Seneca, (2) marks the batch as “acknowledged” and (3) triggers the batch’s commitment at the secondary LU. The commitment copies the data blocks from the secondary log to the secondary LU, deletes the log records, and frees the log space.

Acknowledgements sent to the primary Seneca are used to garbage-collect the primary log: an acknowledged receive batch allows all the records and data blocks in that batch to be deleted from the primary log.

Both the primary and the secondary Seneca can fail during the propagation process. When it recovers, a Seneca instance examines the kind of log it had to determine whether it was the primary or secondary Seneca. If it was a secondary Seneca when it failed, it will re-apply its secondary log up to and including the last acknowledged receive batch, and carry on as a secondary. The situation for a returning Seneca with a primary log is more complicated, which will be discussed in section 3.3.4.

3.3 Seneca state machines

The Seneca protocol is defined in terms of a pair of state machines, one for each of the primary and secondary Seneca modes. Figure 6 shows the state transitions of a Seneca instance in response to failure and recovery events.

When acting as a primary Seneca, the possible failure events are PriLogFilled (primary log filled), PriLogFailed (primary log disk failed), PriFailed (primary Seneca or LU failed), and

SecDisconnected (any failure that leaves the secondary Seneca inaccessible to the primary one, including secondary log disk failure, secondary Seneca LU failure and/or network outage).

The possible recovery events include PriLogReturned (primary log disk is repaired), SecRepaired (secondary returns and contains all the data stored in its log and LU before it crashed) and SecReplaced (secondary returns with empty storage).

Acting as a secondary Seneca, the possible failure events include SecLogFilled (secondary log filled), SecFailed and PriFailed. The possible recovery events include SecRepaired and SecReplaced.

We describe a few interesting corner cases in the Seneca state transitions below.

3.3.1 Failover and fallback

In case of a secondary failure, the primary Seneca continues to perform in the Standalone state, and brings the secondary site back up to date when it returns.

If the primary Seneca fails, the secondary Seneca becomes the new primary. We call this *failover*. When a Seneca returns, it always returns as a secondary Seneca. Even if it was the local Seneca, it needs some preparation, such as cleaning up its log, before it is ready to serve as the primary. When both Senecas are ready, i.e. free of outstanding updates, the local Seneca becomes the primary again. We call this *fallback*.

We assume that some entity outside Seneca will make the decision of whether and when to failover or fallback, and notify Seneca accordingly, because this decision is influenced by many factors outside the storage system, including the state of networks, sites, and applications, as well as the judgement of human operators. It may also involve operations at application level to complete the failover.

Seneca’s failover mode starts when the surviving secondary changes to the Failover state. In the Failover state, the secondary prepares to be the primary by first committing the data in its secondary log to its LU, to bring the LU as up to date as possible. Write requests will be put on hold until Seneca leaves this state, e.g. changes to the Standalone state.

When it is repaired, the secondary Seneca first sends a request to the primary for updates, and then starts to operate in the Normal state. However, if it returns with empty storage, Seneca starts in the DirectUpdate state because the complete snapshot of the primary LU will be propagated and there almost certainly won’t be enough log space for the complete snapshot. After the entire snapshot is committed in the secondary LU, the secondary Seneca changes to the Normal state.

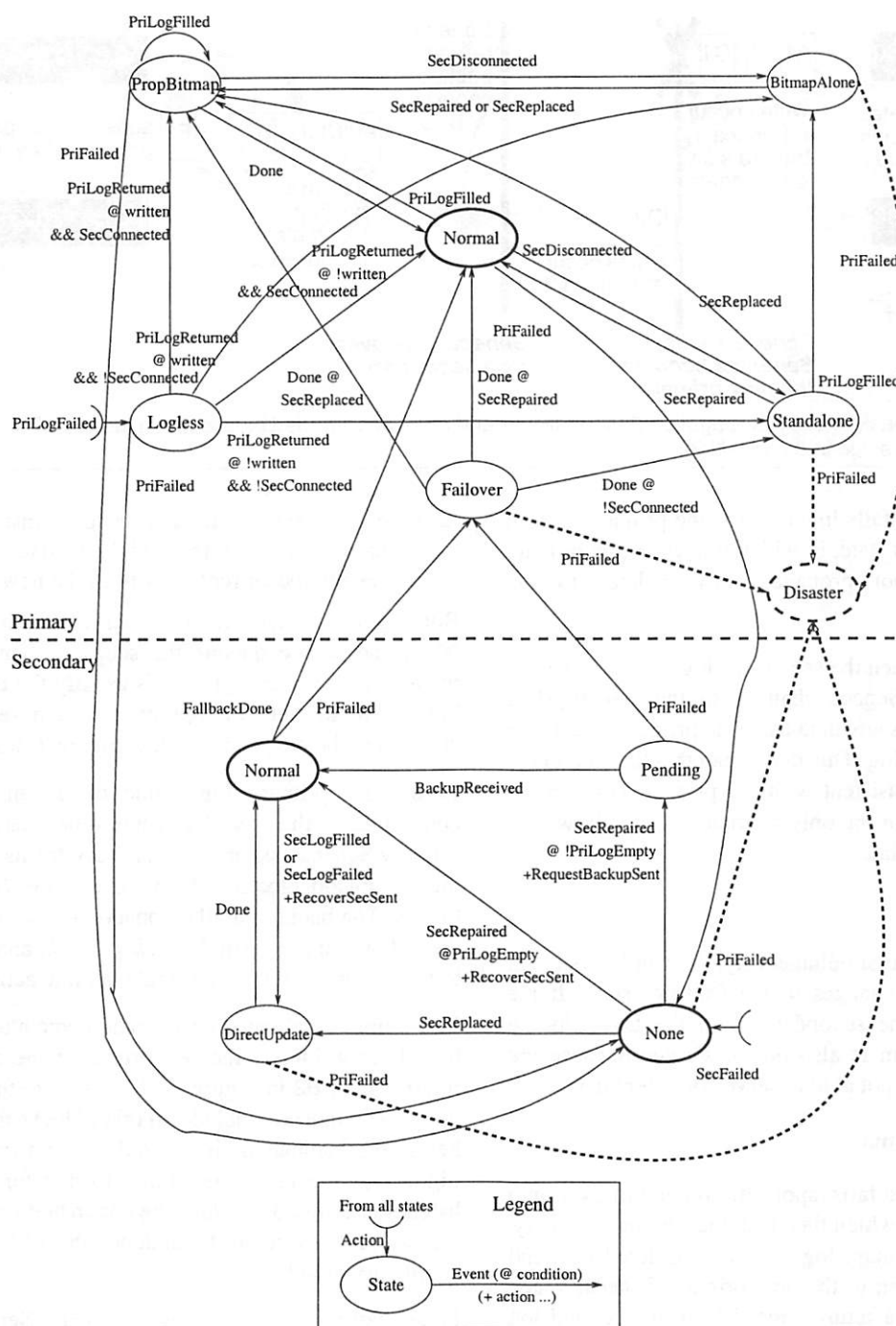


Figure 6: the Seneca state machine. “Primary” and “Secondary” refer to the mode in which a Seneca instance operates.

3.3.2 Log failure

In the Normal state, when the primary log space is filled (e.g., because the secondary Seneca is down for a long enough time, or the network connection to the secondary is too slow), or the log fails, the primary Seneca changes to the PropBitmap state, and merely marks new updates in its bitmap.

In the PropBitmap state, the primary Seneca propagates the blocks in the bitmap (or a complete snapshot of the primary LU if necessary) to the secondary as soon as it can, and asks the secondary to commit all the changes to its LU atomically if possible. However, the secondary may not be able to do so if it is in the DirectUpdate state. As a consequence, the data in the secondary LU could be inconsistent with that in the primary LU until all the changes are committed.

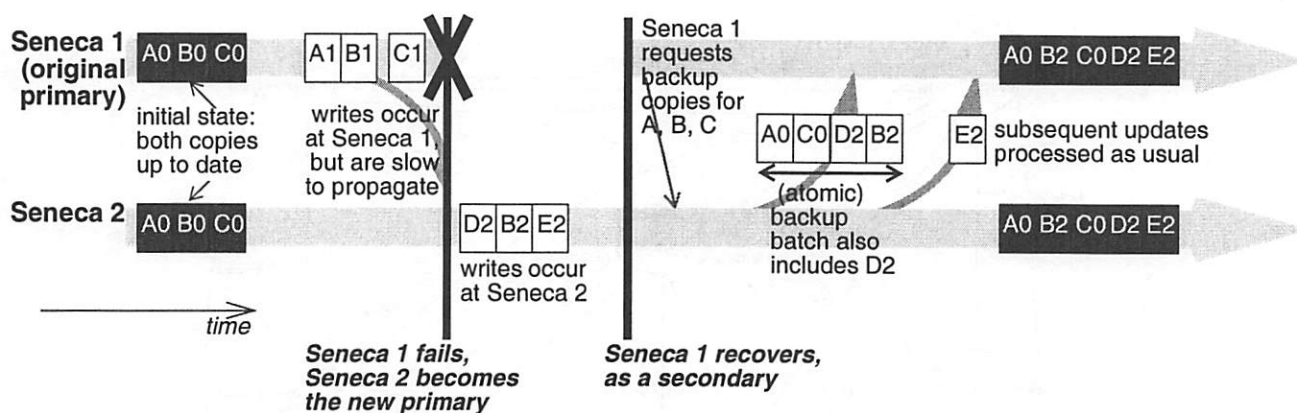


Figure 7: update and backup-copy propagation during failover and recovery. Shaded blocks are in the local LU at each Seneca, unshaded blocks also reside in the local log.

When the primary log fails in any state, the primary Seneca changes to the Logless state, in which it does not record any updates and does not propagate any updates to the secondary.

In the Normal state, when the secondary log space is filled or fails, the secondary Seneca changes to the DirectUpdate state, in which it writes any data from the primary directly to the LU, bypassing the log. This may mean that the data in the secondary LU is inconsistent with the primary copy in the DirectUpdate state—but the only alternative is to allow it to get still further out of date.

3.3.3 Disaster

When both Senecas fail simultaneously, the whole system is out of service, i.e. it changes to the Disaster state. If the primary fails while the secondary is in the DirectUpdate state, the whole system is also out of service because the surviving secondary is not able to serve consistent data.

3.3.4 A failover dilemma

When a primary Seneca fails, applications can fail over onto the secondary Seneca, which then becomes the new primary. Data left in the old primary log will be considered lost, and new data can be written to the new primary Seneca. When the old primary Seneca returns, the data in its LU and log could be inconsistent with that in the new primary Seneca, i.e. both Senecas could have a piece of data that the other does not. The following options exist:

1. Prohibit failover. The entire storage will be unavailable until the old primary Seneca returns, but inconsistency will not occur.
2. Live with inconsistency and minimize data loss. The old primary Seneca will keep the data in its log, and people or applications must resolve the conflicts between the two mirror sites.

3. Live with data loss and eliminate inconsistency. The old primary Seneca will try to undo the data in its log by requesting the current copy from the new primary.

Both options 1 and 2 require intervention at the application level, and so are outside the scope of Seneca's ability to recover. In practice, option 2 is usually the one that is used, but we focus here on option 3, which requires undoing changes in the old primary's log. Figure 7 shows an example.

In the old primary log, some blocks might have been committed to the local LU while others have not. The old primary Seneca asks the new primary for its current copy of the committed blocks, which we call the *backup* of those blocks. The backup of all committed blocks is transmitted to the old primary as a single *backup batch*, and will be written to the old primary LU in a single, atomic action.

In a more complicated case, some committed blocks might have been written in the new primary Seneca before the old returns (e.g., B2 in Figure 7). In this case, the backup batch will be expanded to include all other blocks that were written before the committed blocks in the new primary (e.g. D2 in Figure 7). The reason is similar to that for atomic receive batches (Section 3.1). Once the committed updates at the old primary are successfully undone, the old primary log is simply discarded.

In an extreme case, the new primary Seneca might fail before it sends the backup to the old one. In this case, the old primary Seneca will keep its log and try to commit the logged blocks in it instead. As a consequence, data considered lost in the failover mode (e.g., A1, B1 and C1 in Figure 7) can come back while data written in the failover mode (e.g., D2, B2 and E2 in Figure 7) can disappear after the new primary fails!

3.4 Seneca shadowing

Seneca itself needs to be fault-tolerant; therefore, a local Seneca instance is implemented as a *pair* of Seneca boxes.

The simplest way to achieve this is to have one Seneca box be *active*, while the other acts as a *shadow*, to which the active Seneca can fail over. Changes to the active Seneca's memory are propagated to the shadow before their write operation returns (e.g., in the style of Harp [Liskov1991]). Active-active implementations are also possible.

Ideally, both Seneca boxes will have access to a common log on the SAN, and the only extra latency is the transferring of a short message to the shadowing Seneca across an interconnecting LAN. Therefore, the shadowing Seneca will be a hot standby, i.e., it will have exactly the same state as the primary one when the fail-over occurs. The log disk can be made fault-tolerant by local redundancy schemes, such as RAID5, or mirroring.

3.5 Summary

In this section we presented a remote mirroring protocol in some detail, including the corner cases that make such protocols so challenging in practice.

4 Verifying Seneca correctness

Given the complexity of the corner-cases in a protocol of this form, we felt that it was important to apply some kind of assurance testing to it beyond just prototype and test. Sample approaches include theorem proving, model checking, and simulation. We quickly learned that the Seneca protocols were too complicated to describe at a useful level of detail in any of the languages used for theorem proving and model checking, and cast around for an alternative.

We started from I/O automata [Lynch1989], which model components in asynchronous concurrent systems as labeled transition systems. While the original I/O automata method uses a special purpose language to allow an exhaustive search of the automata state space, our approach was simulation: we built an event-driven simulator that generated events by constrained random walks in the state space, executed the automata, and checked the correctness of the results against our expectations of correct behavior and event sequences. This had the added benefit of allowing the protocol to be implemented in the same language as a real deployment (in our case, C).

It is not our intention here to argue that this approach was the best possible one, but instead to report on our experiences with what turned out to be a fruitful tool.

We wrote automata for each Seneca box, log, LU, WAN and write record, and modelled the Seneca protocol responding to external events. State transitions in the automata take place in response to external events such as write request, log disk failure, Seneca or LU failure and WAN outage. State transitions also take place in response to internal events between the machines such as log space exhaustion, update propagation and batch commit.

We used the model to increase our confidence in:

1. *Coverage*: that Seneca is in a valid state after any sequence of events.
2. *Safety*: that the mirror copies are consistent in normal states, and the sequence of updates at the secondary Seneca is always a prefix of the sequence at the primary.
3. *Liveness*: that data will eventually be written in both mirrors unless a disaster happens.

To check liveness, we stopped the generation of external events and let the automata run until no more internal events were generated. We used assertions inserted in various locations of the model for correctness checking, rather than formal reasoning. We found this approach to be an efficient way of checking a complicated protocol like Seneca, and gaining confidence in it.

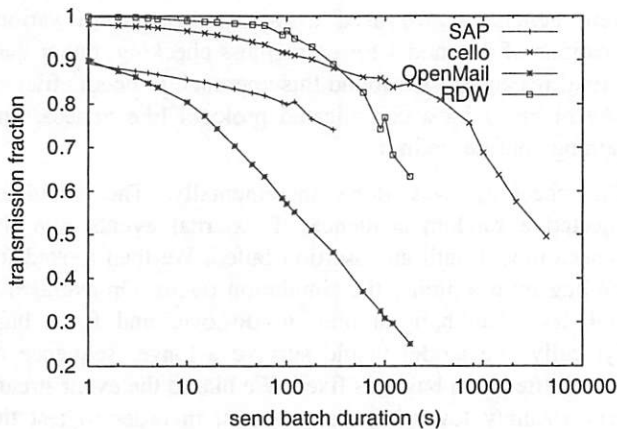
The checking was done incrementally. The simulator injected a random sequence of external events into the Seneca model until an assertion failed. We then looked for the bug by examining the simulation traces. On average, it took less than half an hour to discover and fix a bug. Typically, the model would survive a longer sequence of events after each bug was fixed. We biased the event stream very strongly towards failure events, in order to test the recovery code: typically 15 writes for every external (failure or recovery) event.

Over 131 runs with distinct random seeds, the average count of failure injections before a protocol error was detected was 16435. If we assume that the Seneca's handling of write events is correct, and external failures (such as the failure of a mirrored Seneca or log device) occur at the (relatively high) rate of once a quarter, this corresponds to an estimated mean time between failures (MTBF) of 4100 years for the Seneca protocol proper. Obviously, this is a statistical estimate, not a guarantee, and does not take into account implementation or operator errors—but we still feel that it builds more confidence than merely asserting the protocol's likely correctness.

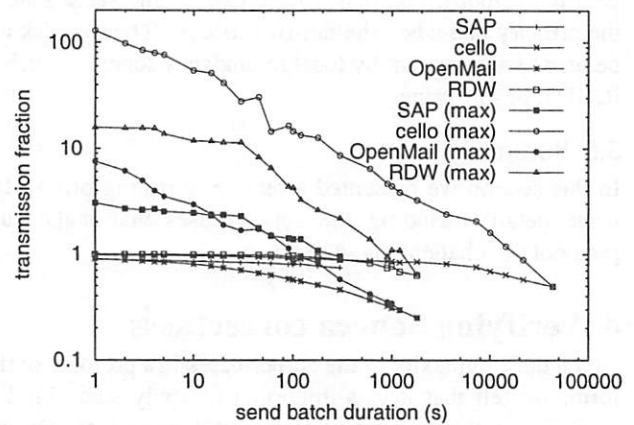
Most of the bugs we found so far were transcription errors in coding the state-transition diagrams we had developed, which would have been too detailed to be covered by a theorem prover or a model checker. In one example, a backup batch (section 3.3.4) included new updates that had already been propagated to and queued in the secondary Seneca, so the queued updates were committed twice, resulting in out-of-order writes to the secondary LU. This kind of error would be hard to detect with traditional testing—especially if failures were induced manually. The last bug we had found at the time of writing was only detected after 1.77M writes, 75.9k failure events and 22.4k recovery events were injected into Seneca, which in turn provoked Seneca to generate an additional 6.60M internal events.

workload	length	date	I/O count	write count	% writes	total data written	mean write rate	peak 1 second write rate
cello2002	24 hours	2002/09/18	6760626	5250126	77.7%	67.40 GB	0.78 MB/s	100.30 MB/s
SAP	15 mins	2002/01/31	4986071	150339	3.0%	1.75 GB	1.95 MB/s	6.75 MB/s
RDW	1.4 hours	2000/02/11	1797210	60758	3.4%	1.70 GB	0.34 MB/s	5.33 MB/s
OpenMail	1 hour	1999/12/17	1287941	931979	72.4%	6.13 GB	1.70 MB/s	15.80 MB/s

Table 2: traced system workload summary



(a) Mean transmission fraction (averaged across all the batches), scaled to the mean write rate.



(b) Largest-batch transmission fraction, scaled to the mean write rate, as well as the mean transmission fraction, for scale. Note the log scale on the Y-axis.

Figure 8: fraction of data transmitted across the WAN, as a function of the batch duration. The left-most point corresponds to no overwrites (for which batch duration is irrelevant); the others are for the indicated batch duration. Both graphs are scaled so that 1.0 corresponds to the mean write rate (see Table 2 for its value).

5 Performance analysis

One of the key questions for Seneca is the amount of network bandwidth that can be saved by delaying I/Os (asynchrony) and coalescing (over-writes). To assess how this will work in practice, we examined a number of real-world I/O traces, summarized in Table 2. The traces were gathered from HP-UX systems, using techniques similar to those used in [Ruemmler1993].

- **cello2002:** an 8-processor HP 9000 N4000 timesharing system for a small group of researchers with 16 GB of RAM, an HP XP512 disk array (total allocated storage was 1.44 TB), running HP-UX 11.00; the trace data is from Wed Sept. 18th, 2002. This is a successor to the 1992 cello system described in [Ruemmler1993].
- **SAP:** an SAP installation from a utility company, running on an HP V2500 that was using SAP ISUCCS 4.5B on top of a 4 TB Oracle database, being accessed by more than 3000 users retrieving customer's utility bills for updating and reviewing. There were also some batch jobs running in the background. The trace was taken in the afternoon of Thursday Jan. 31, 2002. The storage system was an HP XP512 disk array with 160 73 GB disks in

RAID 1/0 mode, a 16 GB cache, and running remote mirroring to a second, remote XP512 via eight ESCON links.

- **RDW:** a retail data warehouse system, containing 500 GB of shopping basket information, executing on an HP V2250 running HP-UX 11.00; the storage system was an EMC Symmetrix array. The trace was made on Tuesday Feb. 22, 2000.
- **OpenMail:** one of 6 HP9000 K580 servers, each of which had 6 CPUs and 3.75 GB of memory and was connected to an EMC Symmetrix 3700 disk array providing 640 GB of storage. The system was running a production OpenMail email workload on HP-UX 10.20 with 4500 users, 1400 of whom were active during the traced period. The trace was taken on Thursday Jan. 20, 2000.

We begin by examining how much the overwrite rate reduces the WAN traffic: quite respectable reductions in total WAN traffic are available: e.g., 5–40% for a batch size of 30 seconds (Figure 8a), and significantly greater reductions for the largest batches (Figure 8b).

Figure 9 confirms that the distribution of batch sizes is far from uniform, which is why the rate-smoothing performed

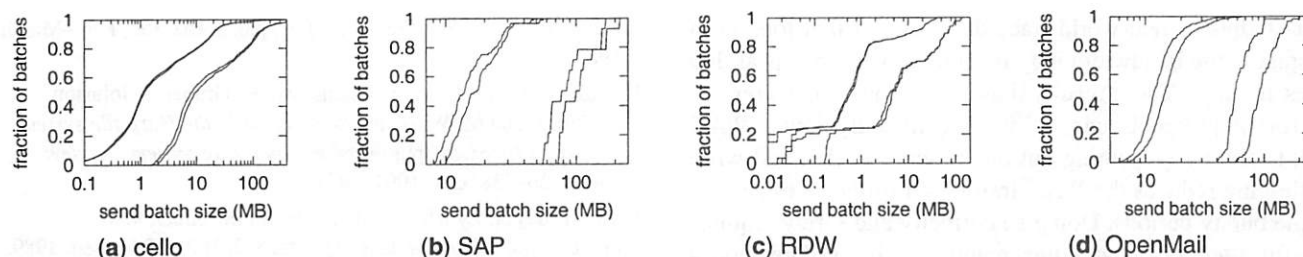


Figure 9: CDFs of measured batch sizes for 10 and 60 second batch durations, with and without overwrites, for the traced systems. The pair of lines to the left are the 10-second case, the leftmost line within each pair is the no-write-coalescing case.

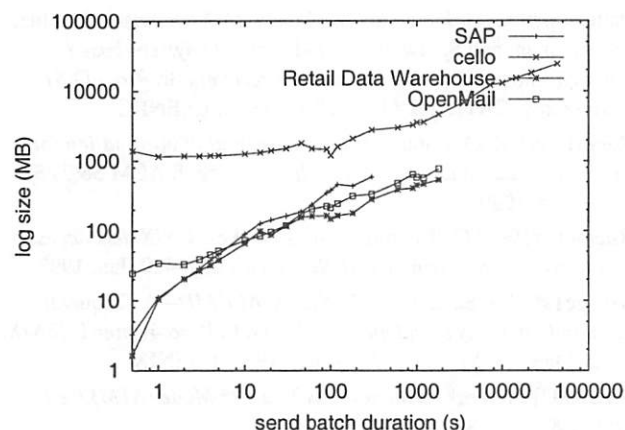


Figure 10: worst-case log size as a function of batch size. The left-most values denote an adaptive-duration batch: the batch end (send barrier) was inserted as soon as the transmission of the previous batch finished.

by asynchronous I/Os is so important. The long-term mean write rate for each of our workloads could be handled by a single 45 Mb/s T3 line, even with no overwrite absorption. But the peak rates require more: the 95th percentile of the 10 second-average I/O rate (Figure 9d), scaled up to cover all 6 OpenMail servers, would require three T3 lines at 85% utilization with write coalescing, and four lines at 93% utilization without.

We determine the *worst-case* log size for each trace, using the 100% T3 link utilization assumption. Figure 10 shows that the log size required to handle write propagation delays is small when the link is up – usually much less than 1 GB. (Cello is an anomaly: it was processing large I/O traces at the time, and its load is a file system, whereas the other systems are databases.) This means that the size of the log is determined almost entirely by the link outage time for which in-order delivery is desired: at these average rates, even a 100 GB log will cover an outage for 14–81 hours.

In some cases, the cost for a WAN link is a function of the data sent over it. To explore the effects of this, we extrapolated the OpenMail workload for all six servers, and applied the cost functions from [SC2002a] to the WAN traffic that resulted. Using no overwrites, the cost would

have been \$40.1k/month; allowing write coalescing reduced the cost to \$31.3k/month with a 1 minute batch duration, or \$12.7k/month with a 5 minute batch duration.

6 Related work

Our goal here is to discuss the genesis/predecessors for the Seneca work itself, rather than existing mirroring products, which were discussed above.

In an analysis of the failure statistics of the Tandem fault-tolerant system [Gray1986a], Jim Gray suggested that remote replication, if one could afford it, protected against 75% of all failures. Partial replication, in the style of RADD [Stonebraker1989], information dispersal [Rabin1989] or Myriad [Chang2002], is probably inappropriate for the applications considered here because of their I/O latencies.

The Jasmin reliable disk server [Uppaluru1987] created a watch dog to copy disk partitions from the active server to the semi-active (once-failed) server, and maintained a water mark to indicate which blocks had been copied and which had not.

Mime [Chao1992], a shadow-writing storage system, avoided synchronous metadata updates using a log and barriers.

The SnapMirror paper [Patterson2002] contains some similar observations to ours, but its focus was on execution performance, rather than the details of the protocol or its correctness; long batch lengths—tens of minutes to hours, rather than seconds; and the support needed in the WAFL no-overwrite file-system for block coalescing, rather than how to minimize the size of the atomic update groups, or the correctness of the resulting protocol.

7 Conclusion

In this paper, we explored the complications and nuances of the remote mirroring problem and the design space of solutions to it, by providing a taxonomy of both. We also described—in some detail—a protocol for asynchronous write propagation that allows safe overwrites, and told of our experiences in validating its correctness and assessing its performance.

Our studies of real-world trace data suggest that long-term average write bandwidth may be quite low, even if peak I/O rates are high. These results show that asynchronous remote mirroring protocols can deliver significantly lower WAN link traffic by smoothing out bursty write traffic, and write coalescing reduces the WAN traffic still further – especially in the bursty periods. Doing so correctly and safely requires careful attention to avoiding many possible failure modes and corner cases. We believe Seneca provides such a protocol.

Acknowledgements

Thanks to David Black and Bob Cochran for helping us understand the intricacies of the remote mirroring space, and to Marvin Theimer, our USENIX shepherd.

References

- [Chang2002] F. Chang, M. Ji, S. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. *Myriad: cost-effective disaster tolerance*. In: *Proc. FAST* (Monterey, CA) pages 103–116, Jan. 2002, USENIX.
- [Chao1992] C. Chao, R. English, D. Jacobson, A. Stepanov and J. Wilkes. *Mime: a high performance parallel storage device with strong recovery guarantees*. HP Laboratories technical report HPL–CSP–92–9rev1 (March 1992, revised November 1992). Available from <http://www.hpl.hp.com/SSP/papers>.
- [de la Croix1975] H. de la Croix and R. G. Tansey. *Gardner's Art Through the Ages*, 6th edition, 1975, Harcourt Grace Jovanovich: New York.
- [EagleRock2001] *Online survey results: 2001 cost of downtime*. Eagle Rock Alliance Ltd. <http://contingencyplanningresearch.com/2001%20Survey.pdf>, accessed Apr. 2002.
- [EMC2002] *Timeline*. EMC Corporation. <http://www.emc.com/ip/timeline.html>, accessed Apr. 2002.
- [EMC–CG2002] *EMC Enterprise SRDF consistency group: description and usage validation*. Engineering white paper, Feb. 2002. EMC Corporation.
- [EMC–SRDF] *Symmetrix Remote Data Facility product description guide*. June 2000. EMC Corporation.
- [EMC–TimeFinder] *EMC TimeFinder product description guide*. Dec. 1998. EMC Corporation.
- [HP–CASA] *hp OpenView continuous access storage appliance*. SAN white paper, Nov. 2002. Hewlett-Packard Company.
- [HP–XP1024] *hp disk array xp1024*. Product brief 5981–0405EN. Apr. 2002. Hewlett-Packard Company.
- [HP–XP–BC] *hp surestore business copy xp*. Product brief 5980–5097EN, 2001. Hewlett-Packard Company.
- [HP–XP–CA] *hp continuous access xp*. Product brief 5980–8608EN, 2001. Hewlett-Packard Company.
- [IBM1997] *DFSMS/MVS Version 1 Remote Copy Administrator's Guide and Reference*. SC35-0169-03, 4th edition, Dec. 1997. IBM Corporation.
- [Kondoff1988] A. Kondoff. *The MPE XL data management system exploiting the HP Precision Architecture for HP's next generation commercial computer system*. Digest of papers, *Spring COMPCON'88* (San Francisco, CA) pages 152–155, Feb.–March 1988. IEEE.
- [Liskov1991] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shriram, and M. Williams. *Replication in the Harp file system*. In *Proc. 13th SOSP*, published as *Operating Systems Review* 25(5):226–238, Oct. 1991. ACM.
- [Lynch1989] N. Lynch and M. Tuttle. *An introduction to input/output automata*. *CWI-Quarterly* 2(3):219–246, Sep. 1989. Centrum voor Wiskunde en Informatica, Amsterdam. Also, Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, MIT.
- [Patterson2002] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. *SnapMirror: file-system-based asynchronous mirroring for disaster recovery*. In *Proc. FAST* (Monterey, CA) pages 117–129 Jan. 2002. USENIX.
- [Rabin1989] M. O. Rabin. *Efficient dispersal of information for security, load balancing, and fault tolerance*. *J. ACM* 36(2):335–348, Apr. 1989.
- [Ruemmler1993] C. Ruemmler and J. Wilkes. *UNIX disk access patterns*. In *Proc. Winter USENIX*, pages 405–420, Jan. 1993.
- [Savage1996] S. Savage and J. Wilkes. *AFRAID—A Frequently Redundant Array of Independent Disks*. In *Proc. Winter USENIX*, (San Diego, CA) pages 27–39, Jan. 1996. USENIX.
- [SBC2002] *FasTrak Asynchronous Transfer Mode (ATM) Cell Relay Service*. SBC Pacific Bell. http://www02.sbc.com/Products_Services/Business/ProdInfo_1/1,,130--1-1-0,00.html, accessed Apr. 2002.
- [SC2002a] *Internet Services: E-RATE master contract number SRC26115*. Sprint Communications Company. Available from South Carolina Division of the State Chief Information Office, <http://www.state.sc.us/oir/rates/docs/sprint-internet-rates.htm>, accessed Apr. 2002.
- [SC2002b] *SCNET ATM POP to POP pricing*. Bell South. http://www.state.sc.us/oir/rates/bellsouth/pdf/SCNET_ATM_POP_TO_POP_RATES.pdf, accessed Apr. 2002.
- [Stonebraker1989] M. Stonebraker, *Distributed RAID – a new multiple copy algorithm*, Technical report UCB/ERL M89/56, Electronics Research Laboratory, University of California at Berkeley, May 1989.
- [Strunk2000] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules and G. R. Ganger. *Self-securing storage: protecting data in compromised systems*. In *Proc. of the 4th OSDI*, pages 165–180. IEEE.
- [Uppaluru1987] P. Uppaluru, W. K. Wilkinson, and H. Lee. *Reliable servers in the JASMIN distributed system*. In *Proc. 7th ICDCS*, pages 105–112, Sept. 1987. IEEE.
- [Veritas2002] *VERITAS Volume Replicator 3.5: Administrator's Guide (Solaris)*. June 2002. Veritas Software Corp.
- [Widen2000] S. Widen. *Compaq VersaStor technology: the road to an open SAN?* IDCFlash #22488, June 2000. International Data Corporation. Available at <http://www.compaq.com/products/storageworks/newsandevents/analyst/22488.html>, accessed Apr. 2002.
- [Witty2001] R. Witty, D. Scott. *Disaster recovery plans and systems are essential*. Gartner FirstTake: FT-14-5021, 12 Sep. 2001. Gartner Research.

Eviction Based Cache Placement for Storage Caches

Zhifeng Chen[†], Yuanyuan Zhou[†], and Kai Li^{*}

[†]*Department of Computer Science
University of Illinois at
Urbana-Champaign
1304 W. Springfield Ave
Urbana, IL 61801*

^{*}*Department of Computer Science
35 Olden Street
Princeton University
Princeton, NJ 08544*

Abstract

Most previous work on buffer cache management uses an access-based placement policy that places a data block into a buffer cache at the block's access time. This paper presents an *eviction-based placement* policy for a storage cache that usually sits in the lower level of a multi-level buffer cache hierarchy and thereby has different access patterns from upper levels. The main idea of the eviction-based placement policy is to delay a block's placement in the cache until it is evicted from the upper level. This paper also presents a method of using a client content tracking table to obtain eviction information from client buffer caches, which can avoid modifying client application source code.

We have evaluated the performance of this eviction-based placement by using both simulations with real-world workloads, and implementations on a storage system connected to a Microsoft SQL server database. Our simulation results show that the eviction-based cache placement has an up to 500% improvement on cache hit ratios over the commonly used access-based placement policy. Our evaluation results using OLTP workloads have demonstrated that the eviction-based cache placement has a speedup of 1.2 on OLTP transaction rates.

1 Introduction

With the ever-widening speed gap between processors and disks, and decreasing memory price, modern high-end storage systems typically have several or even tens of gigabytes of cache RAM [28]. The clients of a storage system, e.g. filers or database servers, also have large amount of devoted main memory for caching [30]. These buffer caches form

a *multi-level buffer cache hierarchy* (See Figure 1). Though the aggregate size of this hierarchy is increasingly larger, the system might not deliver the expected performance commensurate to the aggregate cache size if these caches could not work together effectively. In this paper, we investigate a method to manage the multi-level buffer cache hierarchy effectively. Specifically, we focus on how to make better use of a storage server cache that coexists with large buffer caches of storage clients.

Previous studies [19, 31, 28] have shown that storage caches have different access patterns and thereby should be managed differently from caches at upper level. Accesses to storage caches usually exhibit weak temporal locality because accesses to storage caches are actually misses from upper level buffer caches. In other words, accesses made by applications are first filtered by upper level buffer caches before they reach storage caches. As a result, widely used locality-based cache replacement algorithms, such as Least Recently Used (LRU), do not perform well for storage caches. This has been observed by Muntz and Honeyman's as well as our previous study on file and storage server cache, respectively [19, 31].

Most previous work on file or storage buffer caches focused on cache replacement policies. Buffer cache management mainly consists of two components: replacement policy and placement (admission) policy. A replacement policy decides which block should be replaced to make space for a new block when the cache is full, while a placement policy decides when a block should be brought into a cache. The access-based placement policy has been widely used in most previous studies. This policy places a block into a cache at the time this block is accessed. The main motivation for such a placement is to maintain the *inclusion property* (any block that resides in an upper level buffer cache is also contained in a lower

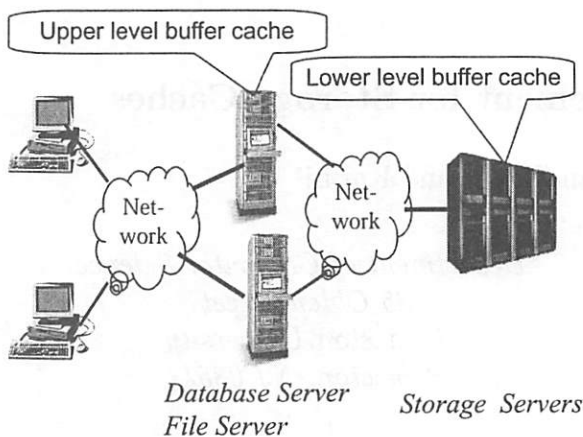


Figure 1: Storage caches in a multi-level buffer cache hierarchy.

level). This property is desirable when the upper level buffer cache is significantly smaller than the lower level one. For example, in a typical main memory hierarchy, the processor cache is several orders of magnitude smaller than the main memory. However, since a storage buffer cache usually has similar size to a storage client cache (e.g., a database or file server cache), maintaining this property at a storage cache has been shown unnecessary and can even hurt performance [19, 31, 28].

To make a storage cache *exclusive*, Wong and Wilkes have proposed an operation called *DEMOTE* to send data evicted from a client buffer to a disk array cache [28]. Their study made a very interesting observation about wasteful inclusiveness of storage caches. They also showed that *DEMOTE* can effectively improve the hit ratios of a storage cache in various simulated workloads. However, to implement this method in a real system, it requires modification to source code of client software, such as a database server, to explicitly utilize this new operation. Besides, in some production systems, the network between a storage client and a storage system can become a bottleneck because of the extra network traffic imposed by *DEMOTE* operations. For applications with intensive I/Os, *DEMOTE* operations can also increase the average miss penalty in a client cache due to waiting for free pages. In addition, the *DEMOTE* method has been evaluated only by simulations, so it is unclear how it would perform in a real system.

This paper generalizes the idea of an exclusive cache and presents an eviction-based cache placement policy for storage caches. Moreover, our method targets application domains where the *DEMOTE* approach is not readily applicable. In par-

ticular, we propose a method called client content tracking (CCT) table to estimate a client's eviction information. This method can avoid modifications to the client's software source code. We also propose letting the storage system decide when to fetch the evicted blocks from disks, a.k.a. *reload*, to avoid delaying demand access at the client side. In other word, our approach is transparent to applications. Since the decision is made by the storage servers, our method also enables more sophisticated optimizations such as eliminating unnecessary reloads or masking reloads using priority-based disk scheduling.

To evaluate the eviction-based placement policy, we have implemented it in both simulators and a storage system connected to a Microsoft SQL server database. Our simulation results with real-world workloads have shown this placement policy can significantly improve cache hit ratios by up to a factor of 5 over the commonly-used access-based placement. Our real system experimental results with OLTP workloads have demonstrated that the eviction-based placement can improve the transaction rate by 20%. We also compared the *DEMOTE* method with our scheme in a storage system. Our implementation results using OLTP workloads have shown that our scheme has a 20% higher transaction rate than the *DEMOTE* method when the client-storage interconnection has a limited bandwidth.

The remainder of this paper is organized as follows. Section 2 briefly describes cache placement policies and metrics to measure those policies. Section 3 presents the benefits of the eviction-based placement policy. Section 4 describes the CCT table to estimate eviction information from a client and Section 5 discusses ways to reduce the reload overheads introduced by the eviction-based placement. After we present the implementation results and the effects of optimizations in Section 6, we conclude the paper and point out the limitations of our study.

2 Cache Placement

A cache placement (admission) policy decides when a missed block should be fetched into a cache. For example, the commonly used access-based placement policy places a block into a cache at the time the block is accessed. The eviction-based placement policy presented in this paper fetches a block into a cache when this block is evicted from an upper level buffer cache.

We use a metric called *idle distance* to evaluate

different cache placement policies. In this section, we describe *idle distance* and then use it to measure the access-based and eviction-based placement policies with three large real-world storage cache access traces.

2.1 Idle distance

To evaluate the effectiveness of different cache placement policies, we need a metric to measure the cost of keeping a block in a cache to generate a cache hit. The idle distance can well serve this purpose. For a reference to a block, its *idle distance* is defined as the period of time this block resides in the cache but is not being accessed. More specifically, for a *reference string* (a numbered sequence of temporally ordered accesses to a cache), we use sequence numbers to denote “time”, and the time of the previous and current access to a block b as $prev(b)$ and $current(b)$, respectively. We then use $place(b)$ to denote the time b is put into the cache. The idle distance for the current reference to b is defined as $current(b) - \max(prev(b), place(b))$, i.e., the time interval from the maximum of b ’s placement time and b ’s previous access time to the current access. During this time interval, b occupies a memory block but is not accessed.

A good cache placement policy should try to reduce idle distance to improve the efficiency of a buffer cache. An ideal policy would put a block into a cache right before it is accessed. But this is impossible unless the system has zero cost to load a missed block.

2.2 Access-based Placement

In the commonly-used access-based placement policy, the *idle distance* for a reference is equal to its *reuse distance*, which is the distance between the previous access and the current access to this block, i.e., $current(b) - prev(b)$. Since the access-based placement policy puts a missed block b into a cache right at its access time, $prev(b)$ equals $place(b)$. Therefore, the reuse distance for this reference is the same as its *idle distance*. Reuse distances have been used by many studies [18, 21, 1, 15] including our previous study [31] to examine the temporal locality in an access sequence.

Our previous study [31] has shown that accesses to storage caches have long reuse distances because accesses from applications have already been filtered through one or more levels of buffer caches before they arrive at storage caches. If a client cache of size k uses a locality based replacement policy like LRU,

after a reference to a block, it takes at least k distinct references to evict this block from the client’s buffer cache. Therefore, the next access to block b in the storage cache is separated by at least k distinct references in the reference sequence at the storage cache. This long reuse distance significantly limits the efficiency of commonly-used access-based placement at storage caches and other lower level buffer caches.

2.3 Eviction-based Placement

In the eviction-based placement policy, the *idle distance* for a reference is equal to its *eviction distance*. At a lower level cache like a storage cache, the eviction distance for a reference is defined as the distance between the current access and the last time it is evicted from a client buffer cache. In other words, if we use $evict(b)$ to denote the “time” when b was most recently evicted from a client buffer cache, the eviction distance for the current access to b is $current(b) - evict(b)$. Since the eviction-based placement policy fetches the block when it is evicted from a client, $place(b)$ equals $evict(b)$. Because an eviction from a client always happens after the previous access to the same block, $prev(b)$ is smaller than $evict(b)$, which implies $\max(prev(b), place(b)) = evict(b)$. Therefore, the idle distance for a reference equals the eviction distance of this reference.

We use idle distance distributions to compare the two placement policies. An idle distance histogram shows the number of references for various distance values. Figure 2 compares idle distance distributions for both access-based placement policy (AC) and eviction-based placement policy (EV) using three real-world storage access traces including:

- **Auspex I/O Trace** is a disk I/O trace collected by filtering the Auspex file Server access trace [6] through an 8 MB NFS file server cache simulator.
- **MS-SQL-Large** is collected from a storage system connecting to a Microsoft SQL database server running the standard TPC-C benchmark [25, 16] for two hours. The TPC-C database contains 256 warehouses and occupies around 100 GBytes of storage excluding log disks. The trace captures all I/O accesses from the Microsoft SQL server to the storage system. The trace ignores accesses to log disks. The Microsoft SQL server cache size is set to be the machine memory limit, 1 Gigabytes.

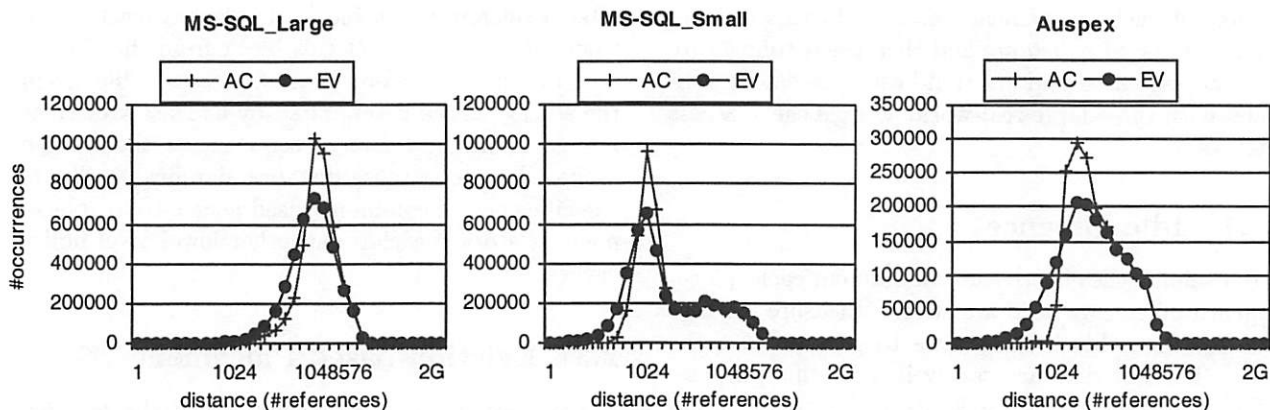


Figure 2: Idle distance distributions for both access-based placement policy (AC) and eviction-based placement policy (EV) with three storage access traces (Note: all figures are in logarithmic scales).

- MS-SQL-Small is collected with the same setup as the previous trace except the database buffer cache size is set to be 64 MBytes. We collected this trace in order to predict results with much larger databases.

As shown on Figure 2, all histogram curves are hill-shaped. Peak distance values, while different, are all relatively large and occur at distances greater than their client cache sizes. This indicates that most of accesses are far apart from previous accesses to the same blocks or previous evictions from clients, showing weaker temporal locality at storage caches.

Comparing the two curves, one can find out that eviction distances are shorter than reuse distances. Figure 2 shows there are fewer occurrences of EV at large distance values (or more occurrences at small distance values) than AC. For example, In the MS-SQL-Large trace, 3.0 million references in AC have idle distances greater than 262144, whereas only 2.3 million references in EV have idle distances greater than 262144. The main reason for this difference is very intuitive. Since a block b first needs to be fetched from a storage cache into a client buffer cache before being evicted from the client cache, $evict(b)$ is usually greater than $prev(b)$. As a result, the eviction distance ($current(b) - evict(b)$) is smaller than the reuse distance ($current(b) - prev(b)$). This implies that the eviction-based placement policy can utilize a storage cache more efficiently than the commonly used access-based placement policy.

3 Benefits of Eviction-based Placement

The eviction-based placement puts a block into a cache when this block is evicted from an upper level cache. This placement policy was first proposed in the victim cache design for hardware processor caches [14]. A victim cache, a small fully-associative cache between a processor cache and its refill path, is used to keep cache blocks that are recently evicted from the processor cache. It has been shown that a victim cache can significantly improve the processor cache performance.

Eviction-based placement is independent from cache replacement policies. Therefore, it can be combined with most replacement algorithms including LRU, Frequency Based Replacement (FBR) [22], 2Q [13], and Multi-Queue (MQ) [31].

To find out the effects of eviction-based placement on cache hit ratios of various replacement policies, we have built four trace-driven cache simulators that respectively use LRU, FBR, 2Q and MQ as the replacement policy. All cache simulators can run with two options: the original (access-based) placement policy and the eviction-based placement policy. Since our first goal is to find out the upper-bound of EV's improvement on hit ratios, we did not simulate disk accesses and network accesses. The extra overheads introduced by EV are discussed in detail in Section 5. These overheads are also reflected in our implementation results on a real system.

Figure 3 compares the hit ratios between the access-based and eviction-based placement policies for four different cache replacements with the MS-SQL-Large trace. LRU + EV means that the cache is managed using LRU as the replacement policy and

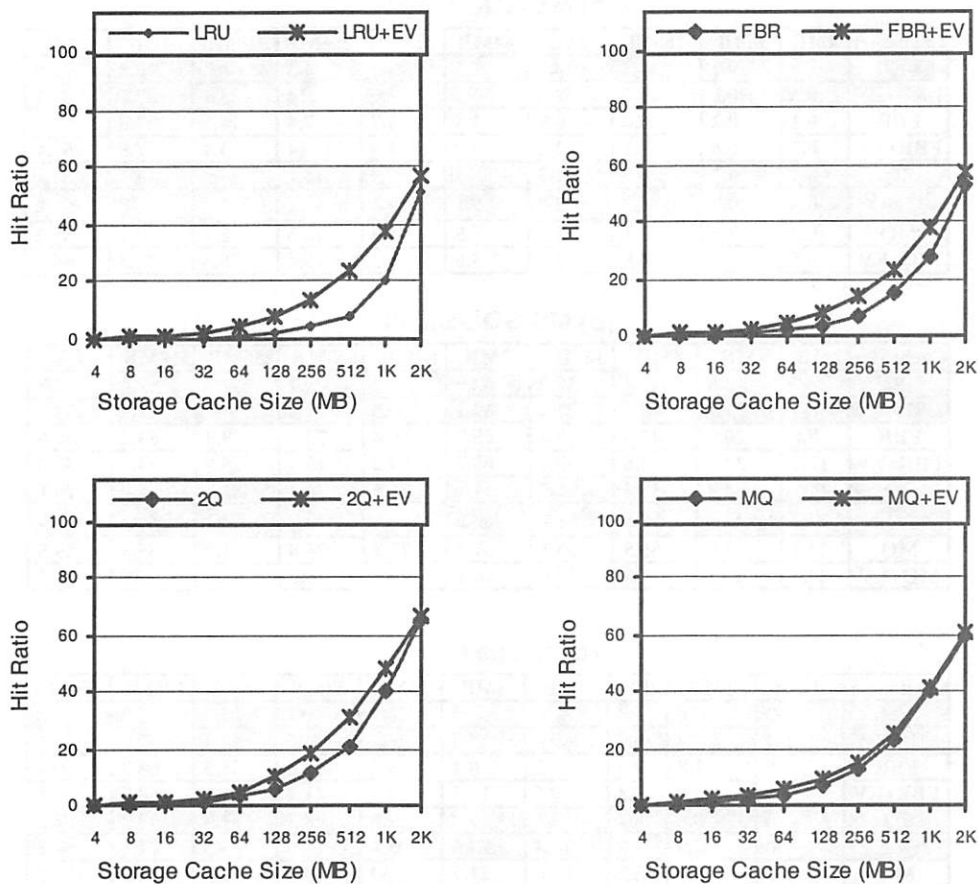


Figure 3: Benefits of eviction-based placement with MS-SQL-Large with different replacement algorithms.

EV as the placement policy, and other abbreviations are similar.

As shown on Figure 3, the eviction-based placement always performs better than the access-based placement. In many cases the gap between these two is quite substantial. For example, when the LRU replacement policy is used, the eviction-based placement has 10% to 5 times higher hit ratios than the access-based placement. The improvements for FBR and 2Q are also significant, up to a factor of 2.

The effects of the eviction-based placement are different for various replacement algorithms. For example, in a 512 MBytes storage cache, the eviction-based placement outperforms the access-based placement by a factor of 2 for LRU, 49% for FBR, 59% for 2Q and only 15% for MQ. The eviction-based placement has the largest improvement on LRU than on the other three replacement algorithms because LRU replaces the block with the longest idle distance from the current time. The idle distance in the eviction-based placement equals the eviction distance, which is always smaller than

the idle distance (reuse distance) in the access-based placement. As a result, some blocks that are evicted by LRU in the access-based placement can stay in the EV-based cache for a longer time to be hit again at next references.

The eviction-based placement has the least impact on MQ among all four replacement algorithms. Since MQ was designed based on the long idle distance access patterns at storage caches, it can selectively keep some frequently accessed blocks in a cache for a longer time. Because of this reason, delaying a block's placement time does not offer large benefit. Therefore, for MQ, EV only has 11-80% improvement over the access-based placement.

The gap between the eviction-based placement and the access-based placement is more pronounced for smaller cache sizes. For example, in the MS-SQL-Large trace with a 128 MBytes storage cache using the 2Q replacement policy, the eviction-based placement has a hit ratio of 9.8% whereas the access-based placement achieves a hit ratio of 5.9%. The gap is even larger for extremely smaller cache size

(a) MS-SQL-Large

CacheSize	4MB	8MB	16MB	32MB	64MB	128MB	256MB	512MB	1GB	2GB
LRU	0	0.1	0.2	0.5	1.1	2.2	4.3	8.2	20.2	51.6
LRU+EV	0.3	0.6	1.1	2.2	4.2	7.9	14	23.4	37.2	57.1
FBR	0.1	0.2	0.4	0.9	1.8	3.7	7.4	14.7	27.8	53.1
FBR+EV	0.3	0.6	1.1	2.2	4.2	7.9	14	23.4	37.8	57.4
2Q	0.1	0.3	0.7	1.5	3	5.9	11.2	20.9	40.4	66.1
2Q+EV	0.3	0.7	1.3	2.6	5.1	9.8	18.1	31.1	48	67.2
MQ	0.3	0.5	1	1.8	3.5	6.6	12.3	22.7	39.1	66.2
MQ+EV	0.5	1	1.8	3.2	5.6	9.3	15.4	25.8	41.3	66.4

(b) MS-SQL-Small

CacheSize	2MB	4MB	8MB	16MB	32MB	64MB	128MB	256MB	512MB	1GB
LRU	3.7	15.5	43.3	57.3	62.5	67	71.7	77.5	82.4	87.8
LRU+EV	16.4	31.2	48.4	57.2	62.1	66.5	71.2	77	82.2	87.6
FBR	8.6	20.3	44.6	60.4	65.7	69.4	73.6	79.2	84.5	88.6
FBR+EV	17.2	32.7	50.6	60.7	65.7	69.2	73.5	78.9	84.1	88.5
2Q	17.1	34.4	54.4	62.2	67.2	71.5	76.3	81.3	86.2	90.1
2Q+EV	27.2	43.4	55.5	61.7	66.6	70.8	75.7	81	86.1	90
MQ	22.1	36.8	55.5	63.1	67.5	72.1	76.8	81.3	85.8	89.5
MQ+EV	23.4	37.7	56	64.6	68.1	73.1	76.6	81.8	86	89.2

(c) Auspex

CacheSize	512KB	1MB	2MB	4MB	8MB	16MB	32MB	64MB	128MB	256MB
LRU	0	0	0	0	2	16.7	36.1	53.3	66.9	78
LRU+EV	1.4	2.8	5.3	9.6	15.8	25.7	40	54.4	67.2	78.1
FBR	0	1.8	2.9	5	8.4	19.2	38.6	55.5	68.3	80.9
FBR+EV	0	2.8	5.3	9.6	16.3	27.3	41.7	56.1	68.7	81
2Q	0	0.6	0.9	1.3	9.4	31.3	48.5	62.8	73.9	84.2
2Q+EV	0	4.3	8	14.4	23.5	35.6	50	63.3	74.1	84.2
MQ	2.3	4.5	8.2	13.6	21.7	33	49.1	62.3	74.5	84.2
MQ+EV	3.1	5.3	8.7	14	21.9	34.2	49.1	63.3	74.8	84.1

Figure 4: Cache hit ratios for all three traces.

(4MBytes), although the hit ratios are so small that two curves in Figure 3 is indistinguishable. But with a 2 GBytes of storage cache, both placement policies have similar cache hit ratios. This can be explained using idle distances. Suppose a storage cache has k blocks. Accesses with idle distances smaller than k can usually hit in the cache, but most of the other accesses would generate cache misses. When k is smaller than the peak idle distance (the distance with most number of references) shown on an idle distance distribution histogram (Figure 2), more accesses have idle distances smaller than k in the eviction-based placement than in the access-based placement. As a result, the eviction-based placement performs better than the access-based placement. But this advantage of eviction-based placement decreases when k increases. As a result, the performance gap between these two also decreases.

Figure 4 shows the hit ratios for all three traces. The overall results for the other two traces are similar to those for MS-SQL-Large. For MQ-SQL-Small, the gap between the two placement policies disap-

pears when the storage cache size is greater than 16 MBytes (2048 8 Kbytes-blocks). This is because the difference in the idle distance distribution between these two policies becomes invisible when the idle distance is greater than 2048 references (see Figure 2).

4 Obtaining Upper Level Eviction Information

Although the eviction-based placement has shown significant benefits over the access-based placement for storage caches, two challenging issues need to be addressed for the eviction-based placement to be used in real systems. The first is to obtain eviction information from client buffer caches. In the hardware victim cache example, when a processor cache evicts a block, it passes the block to the victim cache. However, in most software-managed buffer caches, the eviction information is usually not passed from

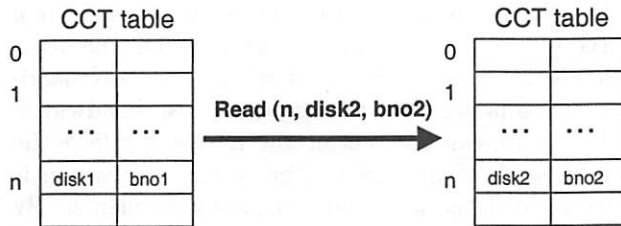


Figure 5: Client Content Tracking Table.

a client to a server. For example, a database buffer cache always silently evicts a clean page and only writes out dirty pages to its back-end storage systems.

Wong and Wilkes [28] have proposed an operation called *DEMOTE* for transferring data ejected from a client buffer cache to a disk array cache. Their approach is similar to the one used in victim caches. Since the current I/O interface between a client and a storage system does not include such an operation, this approach requires modification to client application such as a database server's source code. Therefore, this method is not applicable when the client application source code is not available.

In our study, we use a method that can successfully obtain the client eviction information without any modification to client source code. The main idea is to make use of the buffer address parameter in the I/O read/write interface and build a table to keep track of the contents of the client buffer cache. For example, in a standard I/O interface, a storage I/O read/write call passes at least the following input arguments: disk ID, disk offset, length and buffer address. The buffer address parameter indicates the virtual memory address to store/load the data.

Each entry in the client content tracking (CCT) table records the current disk block (*diskID*, *blockNo*) that resides in each memory location of the client buffer cache. The size of the content table is extensible, i.e., it can grow or shrink dynamically based on the buffer addresses it has seen. Since only 16 bytes are needed for each cache block (of size 8 KBytes in our experiments), the content table does not require too much memory space. For example, if a client uses a 4 GBytes buffer cache, the total memory space needed for a CCT is only 8 MBytes, thereby imposing memory overhead of only 0.2%.

Figure 5 shows a CCT table and how it changes after a read request from a client application. At every read/write operation, CCT is consulted to find out which disk block was previously put in the given client memory address. If the old disk block is differ-

ent from the currently accessed disk block, the old disk block must have been evicted from the client to make space for the new block. Then this eviction information is passed to the storage system. The corresponding CCT entry is modified to point to the currently accessed disk block.

There are two possible places in an I/O subsystem to implement the CCT table: the client side and the storage server side. In our study, we decided to implement it on the client side because it is easier to support clients that use multiple independent storage systems. More specifically, we implement the CCT table in a filter device driver. Since every I/O operation needs to pass through this filter driver, the CCT table can accurately keep track of client buffer cache content. The filter driver can pass eviction information (block numbers) to a corresponding storage node via piggy-backing on read/write messages to that node. Since the driver controls every read/write messages to the storage nodes, it can always find a message to the corresponding node in the send queue to bundle with the eviction information. In this way, no additional message is needed. Because the eviction information is just a few bytes, the additional delay is negligible.

5 Reducing Reload Overhead

The second challenge with the eviction-based placement is to reduce the reload overhead. Since a block's placement into a storage cache is postponed from its access time to the time when it is evicted from the client, the block needs to be reloaded from either clients or disks. As a result, it can increase the network or disk traffic, which can significantly offset the benefits of improved cache hit ratios of the eviction-based placement policy.

The *DEMOTE* mechanism proposed in [28] relies on clients to send an evicted block back to storage systems, even if the block is not dirty. Besides the burden on developers to modify the client software, this method also introduces three performance overheads, which may cancel out the benefits of exclusive caching for some workloads.

- Increased network traffic. *DEMOTE* operations can significantly increase the network traffic from clients to storage systems. As we know, most of the client buffer caches (for example database server buffers) usually try to evict clean pages first before evicting any dirty pages to avoid extra disk write-backs and consistency operations (undo-log logging). In an

OLTP workload, the read traffic is usually 2-3 times larger than the write traffic. If every read request to the storage cache incurs a *DEMOTE* operation, the resulting client-to-server traffic is almost doubled. In a system where the client-storage network is a bottleneck, the *DEMOTE* operations can significantly degrade the system throughput. This has also been pointed out as a limitation of the *DEMOTE* method by the authors themselves [28]. Our implementation results on a storage system also validate this limitation.

- Increased access time. When the buffer cache misses on a client are too bursty to mask the *DEMOTE* overheads, a currently missed block in a client buffer cache may have to wait for a *DEMOTE* operation to finish in order to get a free buffer block before sending a read request to the storage server. Consequently, the average access time will increase in such a case. For example, suppose an application repeats reading sequential blocks from 0 to n in a loop as in a table join operation, where n is larger than the number of blocks in a client buffer cache. Every access would be delayed because it needs to wait for a free block, which is only available after an evicted clean block is sent back to the storage cache.
- Limited flexibility for optimizations. Since a client buffer cache evicts a clean block to make space for a new block, the evicted block needs to be sent to the storage cache before being replaced. Due to this constraint, the time window to demote a block to the storage cache is very short, not enough to perform any effective scheduling or batching optimizations.

In our study, we propose to reload (prefetch) evicted blocks from disks to a storage cache. The first motivation for taking this approach is that the disk bandwidth is usually less utilized than storage area network bandwidth because real-world configurations typically put many disks (for example 60-100 SCSI disks) in a storage server[30]. With an average seek time of 5-6 *ms*, a modern SCSI hard drive can provide over 1MBps bandwidth for a traffic of random 8-KByte block accesses. Thus, without any caching at the storage server, a medium disk array, say 100 disks, can readily saturate a 1Gbps client-storage interconnection. Moreover, a storage server cache can also filter some of the data access traffic. For instance, if a storage cache has a hit ratio of 50%, only half of the network traffic will go to disks.

In this case, using 50 disks per array can saturate a 1Gbps client-storage interconnection. On the other hand, in some environment where the SAN bandwidth is larger than the aggregate disk bandwidth, *DEMOTE* can be a better alternative to relieve the bottleneck of the disks. The second motivation is to avoid delaying demand requests on clients. By pushing reloads to storage systems, client demand requests can proceed without interference by any *DEMOTE* operations.

The third motivation is that one can easily reduce reloading overheads using the following two methods:

(1) **Eliminating unnecessary reloads.** Many reloads in the eviction-based placement are unnecessary. In most cache studies, the rule of thumb is that a large percentage of accesses are made to a small percentage of blocks. This means that most of the blocks (*cold blocks*) are accessed only once or twice in a long period of time. When these blocks are evicted from a client buffer cache, it is unnecessary to reload them from disks. Reloading these blocks can actually degrade the storage cache hit ratios because they can pollute a storage cache. Unfortunately, information on future accesses is usually not available in real systems. In our implementations, we speculate about cold blocks based on the number of previous accesses. In other words, our storage cache does not reload blocks that have been accessed fewer than the *reload_threshold* number of times. This is based on the observation that frequently accessed blocks are more likely to be accessed again in a near future. Many other previous studies [20, 13, 15, 31] were also based on this observation.

(2) **Masking reload overheads through disk scheduling.** To avoid reloads delaying demand disk requests, we give higher priority to demand accesses and lower priority to reloads. We treat reloads in a similar way to prefetching hints since it is perfectly OK if a reload operation is not performed. Given such flexibility, our storage system puts reload operations in a separate task queue and only issues them when there is no ongoing demand request competing for the same disk. Many previous work such as Freeblock scheduling [17] and other scheduling algorithms [8, 23, 11, 29, 2] can easily apply here to mask reload overheads. For example, the reload overheads can be hidden using the Freeblock scheduling that exploits the free bandwidth of disk rotational delay.

6 Evaluation on Real Systems

We implement the eviction-based placement in a storage system using a commercial database server (Microsoft SQL server) as a storage client. The evaluation is conducted using real world OLTP workloads. The goal of our experiments is to answer the following questions.

- How much can the eviction-based placement improve cache hit ratios in real systems?
- What is the overall impact of eviction-based placement on the application performance?
- What are the effects of optimizations for reducing reload overheads?
- What are the tradeoffs between our method and the *DEMOTE* approach [28]?

In this section, we first briefly describe our experimental platform. We then present the performance results, discuss the effects of optimizations and compare our method with the *DEMOTE* method.

6.1 Experimental Platform

We conduct our experiments in a configuration similar to our previous experiments [30]. It consists of three PCs, each of which has dual 933MHz Pentium III Coppermine processors with 256 KBytes L2 cache and 1 GBytes main memory. One PC runs the storage server software, one runs Microsoft SQL server 2000 Enterprise edition, and the last one runs a TPC-C benchmark engine [16] that sends transaction requests to the Microsoft SQL server. The TPC-C benchmark is provided by Microsoft. All PCs use Windows 2000 Advanced Server as operating systems. The TPC-C benchmark requires restoring the database to its initial state before each run to avoid performance discrepancy caused by enlarged database sizes from previous runs. To shorten our experiment execution time, we shrink the number of TPC-C warehouses to 10. The Microsoft SQL server cache size is configured to be 256 MBytes. We run the TPC-C benchmark for 30 minutes in each experiment.

The storage server connects to the database server via a Virtual Interface (VI) network [26] provided by Emulex cLAN network cards. The peak VI bandwidth is about 113 MBps and the one-way latency for a short message is 5.5 μ s. The storage server machine has five Ultra66 IDE disks. The total storage capacity is 200 GBytes. The storage buffer cache size is configured to be 256 Mbytes. The storage

system employs a write-through cache policy. We have implemented both MQ and LRU as the storage cache replacement algorithms. The parameters of the MQ algorithm are set according to our previous study [31]. Our previous study [30] also gives detailed description of the architecture.

6.2 Results Overview

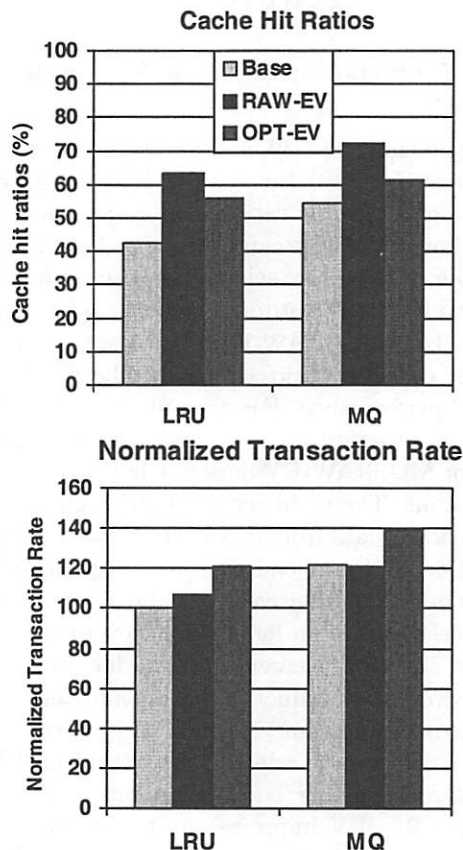


Figure 6: Storage cache hit ratios and normalized transaction rates. All transaction rates are normalized to the ones achieved using the access-based placement and LRU replacement for the storage cache.

Figure 6 compares the storage cache hit ratios and normalized transaction rates for the access-based and the eviction-based placements. We present the results for both LRU and MQ replacements. In these two figures, the base means the access-based placement; RAW-EV means the eviction-based placement without any optimizations; OPT-EV means the eviction-based placement with optimizations to reduce reload overhead.

The raw eviction-based placement has the highest storage cache hit ratios. EV can improve LRU's hit

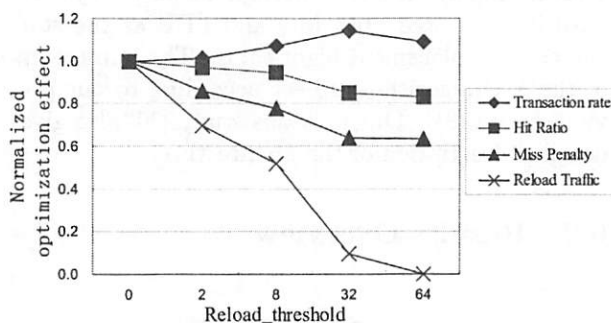


Figure 7: Effects of optimizations for reducing reload overhead.

ratio by a factor of 1.49, and MQ by a factor of 1.32. Similar to the simulation results, EV's improvement on storage cache hit ratios is more pronounced for LRU than for MQ because MQ can better tolerate long idle distances by selectively keeping frequently accessed blocks in a storage cache for a longer time.

Unfortunately, RAW-EV's substantial improvement on cache hit ratios does not fully translate into TPC-C performance. For example, for LRU, RAW-EV only outperforms the access-based placement by 7%. For MQ, RAW-EV does not have any improvement at all. The main reason is the high overheads for reloading data from disks. For reasons explained in Section 5, the reload overheads significantly offset the benefit of improved cache hit ratios. For MQ, the overheads are so large that they totally cancel out the 32% improvement in cache hit ratios.

However, after reducing the reload overheads by eliminating unnecessary reloads and prioritizing demand requests over reloads, the optimized EV can achieve much higher transaction rates. For example, for LRU, EV improves the transaction rate of the base case by a factor of 1.21. For MQ, EV has a speedup of 1.13 over the access-based placement. The effects of optimizations are discussed in detail in the next subsection.

6.3 Effects of Optimizations

To understand the effects of optimizations for reducing reload overheads, we have examined the impact of these optimizations on cache hit ratios, average response time (average miss penalty) of demand disk requests, reload traffic and application transaction rate by varying the *reload_threshold* value. Figure 7 plot these impacts for LRU. All numbers are, respectively, normalized to the ones achieved using RAW-EV. For example, when the *reload_threshold* is 32, the reload traffic is substantially reduced by

OPT-EV to only 0.1 of that with RAW-EV. As a result, the miss penalty decreases to 0.65 of RAW-EV's. Unfortunately, OPT-EV also has a lower cache hit ratio, 0.82 of RAW-EV. Overall, the transaction rate with the optimized version has a factor of 1.13 improvement over RAW-EV when the *reload_threshold* is 32.

When the *reload_threshold* value increases, the number of reloads is significantly reduced, leading to less contention on disks. As a result, the average disk response time for demand requests also decreases. For example, by simply eliminating reloads of blocks that have been previously accessed only once (*reload_threshold* is 2), the reload traffic is reduced by 31%, and the average miss penalty for demand requests is reduce by 14%. The impact on miss penalty is less because some of reloads in RAW-EV are performed when disks are idle, as a result of priority-based scheduling.

However, reducing the number of reloads also has a negative impact. It decreases storage cache hit ratios. For example, increasing the *reload_threshold* value from 0 to 64, the storage cache hit ratio is reduced by 15%. Combining the gain (decrease in disk traffic) and the loss (decrease in cache hit ratios) into the formula: $AverageAccessTime = HitTime * HitRatio + MissPenalty * (1 - HitRatio)$, the impact on application performance varies. The performance peaks when the threshold value is equal to 32.

Notice that our results can be further improved if a more sophisticated priority-based disk scheduling algorithm such as Freeblock scheduling [17] is used to mask reload overheads. We expect the performance improvement with such scheduling algorithm should be similar to the improvement in storage cache hit ratios.

6.4 Comparison with DEMOTE

We also evaluate the tradeoffs between of our method and the *DEMOTE* approach [28]. To do this, we also implement the *DEMOTE* operation in our system. When a clean block is evicted from the storage client (Microsoft SQL server buffer cache in our configuration), the filter driver sends ("demotes") this block to the storage server. Since the database working set size is relative small, we vary the database-storage network bandwidth in a range from 40MB/s to 113MB/s. Since the VI network in our platform can provide 113 MB/s user-to-user bandwidth, we have to run a simple ping-pong VI test program on the side to generate network traffic to utilize 1/3 or 2/3 of the VI bandwidth. The test

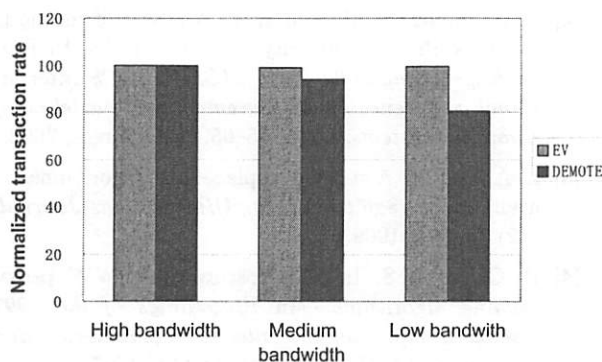


Figure 8: Normalized transaction rates of the EV and *DEMOTE* placement policies under three different configurations of network bandwidth. All transaction rates are normalized to their corresponding rate with 113MB/s network bandwidth.

program is very simple and introduces little processor overhead.

Figure 8 compares the performance of *DEMOTE* and EV under three different configurations of network bandwidth. When the available client-storage network bandwidth is high compared to the client workloads, *DEMOTE* and EV perform similarly. However, when the network bandwidth is low, EV outperforms *DEMOTE* by 20%, even though both approaches have similar cache hit ratios. This is because EV does not impose extra network traffic, whereas *DEMOTE* can potentially double the traffic. This result of *DEMOTE* in a real system matches the simulation result. These results indicate that EV would be a better alternative when the client-storage network has limited bandwidth.

7 Related Work

A large body of literature has examined the cache management problem. However, most previous work has focused on varying cache replacement policy with a fixed placement policy—access-based placement policy. The classic buffer replacement algorithms include the Least Recently Used (LRU) [9, 5], First in First Out (FIFO), Most Recently Used (MRU) and Least Frequently Used (LFU). Recently proposed algorithms include FBR [22], LRU-k [20], 2Q [13], LFRU [15], MQ [31], LIRS [12], and *DEMOTE* [28], just to name a few. These algorithms have shown performance improvement over the widely used LRU algorithm for evaluated workloads. In the spectrum of off-line algorithms, Belady's OPT algorithm and WORST algorithm [3, 18] are widely used to derive a lower and upper bound

on the cache miss ratio for replacement algorithms.

Our work is motivated by previous studies [7, 10, 19, 31, 27, 28, 4]. Dan, Dias and Yu conducted a theoretical analysis of hierarchical buffering in a shared database environment [7]. Franklin, Carey and Livny also explored global memory management in database systems [10]. Muntz and Honeyman investigated multi-level caching in a distributed file system, showing that server caches have poor hit ratios [19]. Willick, Eager and Bunt have demonstrated that the Frequency Based Replacement (FBR) algorithm performs better for file server caches than locality based replacement algorithms such as LRU [27]. Cao and Irani showed that GreedyDualSize replacement algorithm performs better than other known policies for a web cache [4]. Storage caches had been shown to exhibit different access pattern and thereby should be managed differently from other single level buffer caches [31].

The eviction-based placement was first proposed in the hardware victim cache work [14] for hardware processor caches. However, few software managed buffer caches have used eviction-based placement because upper level buffer caches usually do not provide any eviction information to lower level caches. Wong and Wilkes proposed a *DEMOTE* operation to transfer data evicted from the client buffer to the storage cache [28]. This work has made a very good observation, i.e., storage caches should be made exclusive. Their simulation evaluation has shown promising results. But their approach has some limitations as described in previous sections. This study generalizes their approach and proposes alternatives to address those limitations. Moreover, we also evaluate the *DEMOTE* method in a storage system.

Many past studies have used metrics such as LRU stack distance [18], marginal distribution of stack distances [1], distance string models [24], interference gap (IRG) model [21] or temporal distance distribution [31] to analyze the temporal locality of programs. In our study, we use idle distance distributions to measure the effects of cache placement policies.

8 Conclusions

This paper presents an eviction-based cache placement policy to manage storage caches. This placement policy puts a block into a storage cache when it is evicted from a client buffer cache. We have also described a method of using a client content

tracking table to obtain eviction information from client buffer caches without modifying client applications. To reduce the reloading overheads introduced by the eviction-based placement, we have discussed two techniques, eliminating unnecessary reloads and masking reloads using priority-based disk scheduling.

Our simulation results of real-world workloads show that the eviction-based cache placement has 10% to 500% higher cache hit ratios than the access-based placement policy for four different cache replacement algorithms. Our implementation results on a storage system connected to Microsoft SQL server with OLTP workloads have demonstrated that the eviction-based cache placement can improve the application transaction rate by 20%. We also compare our method with *DEMOTE* in a storage system. Our implementation results show that our method has a 20% higher transaction rate than the *DEMOTE* method when the client-storage network has limited bandwidth.

This paper has several limitations. First, we have only used some simple techniques to reduce reloading overheads. We are currently implementing the Freeblock scheduling [17] to mask reloading overheads. Second, we have not done theoretical analysis on the eviction-based placement policy. Some theoretical analysis would be useful to better understand the characteristics of different cache placement policies. Third, we have studied only two types of storage workloads: one is a database OLTP workload and the other is a file system workload. It is interesting to see how well the eviction-based placement would work for other workloads. Even though this paper focuses on storage cache management, the techniques presented in this paper can easily apply to other lower level buffer cache management.

9 Acknowledgement

The authors are grateful to our shepherd Marvin Theimer for his invaluable feedback in preparing the final version of the paper. We would also like to thank anonymous reviewers for their detailed comments.

References

- [1] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS)*, Dec. 1996.
- [2] E. Bachmat and J. Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 55–65. ACM Press, 2002.
- [3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
- [4] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.
- [5] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice Hall, Englewood Cliffs, 1973.
- [6] M. D. Dahlin, C. J. Mather, R. Y. Wang, T. E. Anderson, and D. A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 150–160, 1994.
- [7] A. Dan, D. M. Dias, and P. S. Yu. Analytical modelling of a hierarchical buffer for a data sharing environment. *ACM SIGMETRICS Performance Evaluation Review*, 19(1):156–167, May 1991.
- [8] P. Denning. Effects of scheduling on file memory operations. In *AFIPS Spring Joint Computer Conference*, volume 31, pages 9–21, Washington, D.C., 1967. Thompson Book Co.
- [9] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.
- [10] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *International Conference On Very Large Data Bases (VLDB '92)*, pages 596–609, San Mateo, Ca., USA, Aug. 1992. Morgan Kaufmann Publishers, Inc.
- [11] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7rev1, Hewlett-Packard Company, February 1991.
- [12] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the international conference on Measurement and modeling of computer systems*, pages 31–42. ACM Press, 2002.
- [13] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 439–450, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

- [14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 364–373. ACM Press, May 1990.
- [15] D. Lee, J. Choi, J.-H. Kim, S. L. Min, Y. Cho, C. S. Kim, and S. H. Noh. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, volume 27,1 of *SIGMETRICS Performance Evaluation Review*, pages 134–143, New York, May 1–4 1999. ACM Press.
- [16] S. T. Leutenegger and D. Dias. A modeling study of the TPC-C benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):22–31, June 1993.
- [17] C. Lumb, J. Schindler, G. R. Ganger, E. Riedel, and D. F. Nagle. Towards higher disk head utilization: Extracting “free” bandwidth from busy disk drives. In *Symposium on Operating Systems Design and Implementation*, 2000.
- [18] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
- [19] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems -or- your cache ain't nuthin' but trash. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 305–314, Berkeley, CA, USA, Jan. 1991. Usenix Association.
- [20] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 297–306, Washington, D.C., 26–28 May 1993.
- [21] B. G. V. Phalke. An inter-reference gap model for temporal locality in program behavior. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 291–300, New York, NY, USA, May 1995. ACM Press.
- [22] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 134–142. ACM Press, 1990.
- [23] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, CA, 1990. USENIX Association.
- [24] J. R. Spirn. Distance string models for program behavior. *Computer*, 9(11):14–20, Nov. 1976.
- [25] Transaction Processing Performance Council. *TPC Benchmark C*. Shanley Public Relations, 777 N. First Street, Suite 600, San Jose, CA 95112-6311, May 1991.
- [26] Virtual interface architecture specification version 1.0. VI-Architecture Organization, 1997.
- [27] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network file servers. In *13th International Conference on Distributed Computing Systems*, Pittsburgh Hilton, Pittsburgh, PA, May 1993. IEEE. University of Saskatchewan, Canada.
- [28] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proc. of the USENIX Annual Technical Conference*, pages 161–175, 2002.
- [29] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 241–251. ACM Press, 1994.
- [30] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li. Experiences with VI communication for database storage. In *the 29th International Symposium on Computer Architecture (ISCA)*, May 2002.
- [31] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proc. of the USENIX Annual Technical Conference*, June 2001.

Fast, Scalable Disk Imaging with Frisbee

Mike Hibler Leigh Stoller Jay Lepreau Robert Ricci Chad Barb

*School of Computing
University of Utah*

Abstract

Both researchers and operators of production systems are frequently faced with the need to manipulate entire disk images. Convenient and fast tools for saving, transferring, and installing entire disk images make disaster recovery, operating system installation, and many other tasks significantly easier. In a research environment, making such tools available to users greatly encourages experimentation.

We present Frisbee, a system for saving, transferring, and installing entire disk images, whose goals are speed and scalability in a LAN environment. Among the techniques Frisbee uses are an appropriately-adapted method of filesystem-aware compression, a custom application-level reliable multicast protocol, and flexible application-level framing. This design results in a system which can rapidly and reliably distribute a disk image to many clients simultaneously. For example, Frisbee can write a total of 50 gigabytes of data to 80 disks in 34 seconds on commodity PC hardware. We describe Frisbee's design and implementation, review important design decisions, and evaluate its performance.

1 Introduction

While most computer use focuses on creating, storing and moving single files, many application domains exist where efficiently handling operations on entire disks is important. Classic system administration tasks, such as operating system installation, catastrophe recovery, and forensics, as well as new research such as mobile work environments [18] and computing utility farms [9, 14], benefit greatly from the ability to quickly read, transfer, and write entire disk partitions.

There are two basic disk-level distribution strategies. Differential update, represented by tools such as `rsync` [17] operates above the filesystem, and compares what is already on the disk with the desired contents, replacing only what is necessary. Disk imaging, used by

tools such as Ghost [7], operates below the filesystem, unconditionally replacing the contents of a disk.

Differential techniques are very effective at synchronizing file hierarchies within a filesystem and are extremely bandwidth efficient. However, for distributing entire disks, disk imaging offers a number of important advantages:

Generality: Creating a disk image requires no knowledge of the filesystem being imaged. We show how limited knowledge can be beneficial in Section 3.1, but is not required. Synchronization above the filesystem, however, requires detailed understanding of the filesystem such as directory structure, file ownership, access controls, and times.

Robustness: Disk images have no dependence on the existing contents of the target disk; in contrast, file-based synchronization tools cannot, for example, be used on a corrupted filesystem.

Versatility: One filesystem type can easily be replaced with another using a disk image. This cannot be done with a file-based synchronizer.

Speed: Writing an entire disk image can be faster than determining which files need to be updated. Section 5.4 demonstrates that Frisbee runs much faster than `rsync` in our target environment.

Full-disk imaging does have drawbacks. It is less bandwidth-efficient than differential techniques—no matter how small the difference between the source and destination, the entire disk is copied. The client will also most likely have to be entirely dedicated to the task, since instead of updating files above the filesystem layer, raw disk contents are changed. The advantages outweigh the drawbacks, however, and we demonstrate that by taking advantage of characteristics of the target domain, disk imaging can be used to implement an efficient and scalable disk loading system.

Adopting the strategy of complete disk reloading due to our application's requirements, we have designed, implemented and evolved Frisbee, a disk image generation and distribution system that is fast and highly scalable in a LAN environment. While designed for our network emulation testbed, Frisbee offers functionality and techniques useful in a variety of production and research environments.

Five aspects of Frisbee's design are key to its success:

This work was largely sponsored by NSF grants ANI-0082493 and ANI-0205702, Cisco Systems, and DARPA grant F30602-99-1-0503.

Author information: {mike,stoller,lepreau,ricci,barb}@cs.utah.edu, School of Computing, 50 S. Central Campus Drive, Rm. 3190, University of Utah, Salt Lake City, UT 84112-9205.

www.flux.utah.edu www.emulab.net www.netbed.org

- Domain-specific data compression. Although Frisbee's overall approach treats disk contents as opaque, we relax this requirement for several common filesystem types, building in enough knowledge of their formats to enable identification of free blocks and entirely avoid distributing or writing them.
- Two-level segmenting of the data, into randomly-accessed long segments composed of small blocks typically accessed sequentially. This approach meets the many competing needs of disk I/O, block decompression, network transmission, selective retransmission, and relative simplicity.
- A custom receiver-driven reliable multicast protocol, optimized for the LAN environment.
- Careful concurrency control in the receiver, using three-way multithreading to fully overlap I/O with decompression.
- Designing for this domain's particular pattern of resource availability: target machines entirely dedicated to the diskloading task, a shared server, and a secure high-bandwidth local network.

This paper makes the following contributions: (1) It shows that bulk disk imaging can be extremely fast and scalable, making it a practical approach to disk loading, and frequently a superior one. Furthermore, our performance results indicate that disk imaging is so fast that it can be applied in qualitatively new ways. (2) It presents the detailed design, implementation, and experimental evaluation of Frisbee and identifies the design aspects most important to its performance. (3) The disk imaging system it describes is, to our knowledge, the fastest such system extant. In addition, versions of the system have been proven in production use for over 18 months by hundreds of external users, and is available in open source form. (4) It discusses the design tradeoffs in disk imaging systems.

In the rest of this paper we first outline Netbed, the network testbed system that drove our need for disk imaging. We then outline the design tradeoffs in a disk imaging system, following with sections on Frisbee's design and implementation, performance evaluation and analysis, related and future work, and conclusion.

2 The Netbed Context

As Frisbee was developed primarily for use in the Emulab portion of Netbed, some background will be useful in understanding its motivation and target environment. Netbed [20] is a time- and space-shared facility for networking and distributed systems research and education. It has been in use by the community since April 2000. Emulab is one of Netbed's primary hardware resources—it consists of a cluster of commodity

PC "nodes" with configurable network interconnectivity. Emulab is space-shared in the sense that it can be arbitrarily partitioned for use by multiple experimenters at the same time. Some resources in the system, such as nodes, can only be used in one experiment at a time; in this sense, Emulab is also time-shared. Experimenters are generally given full root access to nodes, meaning that they are free to reconfigure the host operating system in any way they wish, or even install their own.

In light of its time-shared nature, and the degree of control experimenters are given, it is critical that Emulab nodes be returned to a known state between experiments. Without this, experimenters have no guarantee that their results are not affected by configuration changes made by the previous user. Even worse, a malicious user could modify disk contents to facilitate compromise of the node once it has been allocated to another experimenter. To ensure a node is in a known state, its disk must be entirely reloaded.

Disk contents on Emulab are considered to be soft state—hard state, such as user accounts and information about network configuration, are kept off-node in a central database. This separation simplifies things by allowing the same disk image to be used on many nodes, with each node self-configuring from the central database at boot time. It also relieves users of the need to preserve configuration data. Thus, if an experimenter corrupts disk contents, say by introducing a kernel change that corrupts a filesystem, the disk can simply be reloaded, preserving any hard state and losing only soft state. This forgiving environment encourages aggressive experimentation.

Disk images can also be loaded automatically at experiment creation time. An experimenter who wishes to install their own custom operating system or make substantial changes to the default FreeBSD or Linux images provided by Emulab can create an image containing their customizations. They can then load this image on an arbitrary number of other nodes without manual intervention.

Since Emulab has a large number of nodes (currently, 170), and users have run experiments that use more than 120 nodes, speed and scaling are critical to enable these new uses of disk imaging. Waiting for scores of nodes to load serially would leave them unavailable for a long period of time, dramatically reducing the throughput of Emulab, so image distribution must be done in parallel. There are, however, distinct classes of nodes of differing speeds, so it is important that our disk imaging solution work well with heterogeneous clients.

3 Design Tradeoffs

There are three phases of a disk imaging system: image creation, image distribution, and image installation. Each phase has aspects which must be balanced to fulfill a desired goal. We consider each phase in turn.

3.1 Image Creation

In image creation, the goal is to create a consistent snapshot of a disk or partition in the most efficient way possible.

Source availability: While it is possible for the source of the snapshot to be active during the image creation process, it is more common that it be quiescent to ensure consistency. Quiescence may be achieved either by using a separate partition or disk for the image source or by running the image creation tool in a standalone environment which doesn't use the source partition. Whatever the technique, the time that the image source is "offline" may be a consideration. For example, an image creation tool which compresses the data as it reads it from the disk may take much longer than one that just reads the raw data and compresses later. However, the former will require much less space to store the initial image.

Degree of compression and data segmentation: Another factor is how much (if any) and what kind of compression is used when creating the image. While compression would seem to be an obvious optimization, there are trade-offs. As mentioned, the time and CPU resources required to create an image are greater when compressing. Compression also impacts the distribution and decompression process. If a disk image is compressed as a single unit and even a single byte is lost during distribution, the decompression process will stall until the byte is acquired successfully. Thus, depending on the distribution medium, images may need to be broken into smaller pieces, each of which is compressed independently. This can make image distribution more robust and image installation more efficient at the expense of sub-optimal compression.

Filesystem-aware compression: A stated advantage of disk imaging over techniques that operate at the file level is that imaging requires no knowledge of the contents or semantics of the data being imaged. This matches well with typical file compression tools and algorithms which are likewise ignorant of the data being compressed. However, most disk images contain filesystems and most filesystems have a large amount of available (free) space in them, space that will dutifully be compressed even though the contents are irrelevant. Thus, the trade-off for being able to handle any content is wasted time and space creating the image and wasted time decompressing the image. One common workaround is to zero all the free space in filesystems

on the disk prior to imaging, for example, by creating and then deleting a large file full of zeros. This at least ensures maximum compressibility of the free space. A better solution is to perform *filesystem-aware* compression. A filesystem-aware compression tool understands the layout of a disk, identifying filesystems and differentiating the important, allocated blocks from the unimportant, free blocks. The allocated blocks are compressed while the free blocks are skipped. Of course, a disk imaging tool using filesystem-aware compression requires even more intimate knowledge of a filesystem than a file-level tool, but the imaging tool need not understand all filesystems it may encounter—it can always fall back on naive compression.

3.2 Image Distribution

Image distribution is concerned with getting a disk image from a "server" to one or more "clients." In our context it is assumed that the server and clients are different machines and not just different disks on the same machine. Furthermore, we restrict the discussion to distribution over a network.

Network bandwidth and latency: Perhaps the most important aspect of network distribution is bandwidth utilization. The availability of bandwidth affects how images are created (the degree of compression) as well as how many clients can be supported by a server (scaling). Bandwidth requirements are reduced significantly by using compression. Increased compression not only reduces the amount of data that needs to be transferred, it also slows the consumption rate of the client due to the need to decompress the data before writing it to disk. If image distribution is serialized, only one client at a time, then compression alone may be sufficient to achieve a target bandwidth. However, if the goal is to distribute an image to multiple clients simultaneously, then typical unicast protocols will need to be replaced with broadcast or multicast. Broadcast works well in environments where all clients in the broadcast domain are involved in the image distribution. If the network is shared, then multicast is more appropriate, ensuring that unaffiliated machines are not affected. Just as in all data transfer protocols, the delay-bandwidth product affects how much data needs to be en route in order to keep clients busy, and the bandwidth and latency influence the granularity of the error recovery protocol.

Network reliability: As alluded to earlier, the error rate of the network may affect how compression is performed. Smaller compression units may limit the effectiveness of the compression, but increase the ability of clients to remain busy in the face of lost packets. More generally, in lossy networks it is desirable to subdivide an image into "chunks" and include with each chunk additional information to make that chunk self-

describing. In a highly reliable network, or if using a reliable transport protocol that provides in-order delivery beneath the image distribution protocol, this additional overhead would be unnecessary.

Network security: If the distribution network is not “secure,” additional measures will need to be taken to ensure the integrity and privacy of image data. If the image contains sensitive data, then encryption can be used to protect it. This encryption can be done either in the network transport using, for example, SSL, or the image itself could be encrypted as part of the creation process. The latter requires more CPU resources when creating the image but provides privacy of the stored image and is compatible with existing multicast protocols. Ensuring that the image is not corrupted during distribution due to injection of forged data into the communication channel is also an issue. This requires that clients authenticate the source of the image. Again, many solutions exist in the unicast space, such as using an SSH tunnel to distribute images. For multicast, the problem is harder and the focus of much research [2]. Note that security is not just a wide-area network concern. Even in a LAN, untrusted parties may be able to snoop or spoof on traffic unless countermeasures are taken. However, in the LAN case, switch technologies such as virtual LANs can provide some or all of the necessary protection.

Receiver vs. sender-driven protocol: A final issue in image distribution is whether the protocol is server or receiver-driven [19]. A simple server-driven protocol might require that all clients synchronize their startup and operate in lock-step as the server doles out pieces of an image as it sees fit. Such a strategy would scale well in a highly reliable network with homogeneous client machines as little extraneous communication is required. However, if a client does miss a piece of the image for any reason, it might be forced to abort or wait until the entire image has been sent out and then request a resend. A client-driven protocol allows each client to join the distribution process at any time, requesting the chunks it needs to complete its copy, and then leaving. The process completes when all clients have left. The downside is more control traffic and the potential for significant redundant data transfers, either of which can affect scaling.

3.3 Image Installation

The final element of disk imaging is the installation of the transferred image on a client. As with image creation, the disk or partition involved must be quiescent with the image installer either operating on a second disk or partition or running standalone. Since image installation is typically concerned with installing or restoring the “primary” disk on a machine, we restrict the remaining discussion to the standalone case.

Resource utilization: In a standalone environment, the disk installation tool is in the enviable position of being able to consume every available local resource on the target machine. For example, it can use hundreds of megabytes of memory for caching image data incoming from the network or for decompressed data waiting to be written to disk. Likewise, it can spawn multiple threads to handle separate tasks and maximize overlap of CPU and I/O operations.

Overlapping computation and I/O: CPU is of particular interest since, on reasonably fast current processors, substantial computing can be performed while waiting for incoming network packets and disk write completions. The most obvious use of the cycles is to decompress data. However, on unreliable transports the time could also be used for computing checksums, CRCs, or forward-error-correction codes. On insecure transports, CPU resources may be needed for decrypting and authenticating incoming data.

Optimizing disk I/O: If disk I/O is the bottleneck when installing an image, the installation tool may be able to exploit client resources or characteristics of the disk image to minimize disk write operations. On machines with large physical memories, memory can be used to buffer disk writes allowing for fewer and larger sequential IO operations. If the image format uses a filesystem-aware compression strategy which distinguishes allocated and free blocks, the installation tool can seek over ranges of free blocks, thereby reducing the number of disk writes. Note that this method has security implications, since it has the potential to “leak” information from the previous disk user to the new user.

Optimizing network I/O: If network bandwidth is the bottleneck then it may be possible to take advantage of similarities between the old disk contents and the desired contents, as is done in the LBFS filesystem [15], designed for the wide-area. In this technique, acquiring a disk image would be a two-phase process. Whenever a client needs a block of data, it would first ask for a unique identifier, such as a collision-resistant hash [5], for that block which it could then compare to blocks on the local disk. If the block already exists on the local disk, it need only be copied to the correct location. Only if the block is not found, would the client request the actual block data. Such hashing techniques can place a heavy burden on the CPU as well as the disk, if local hashes must be computed at run time.

4 Design and Implementation

The previous section outlined a variety of issues and trade-offs in the design of a disk imaging system. In this section we describe our design choices, their rationale, and their mapping to implementation.

4.1 Overview

Creation and compression: To create an image, we first boot the source machine into a special memory file system-based version of Unix. This satisfies the need for disk quiescence, and allows us to create images without porting the image creator to run on all operating systems for which it can create images. Since image creation is much less frequent than image installation, we do not aggressively optimize the time spent creating images. To save on server disk space and bandwidth, the image is compressed on the client before being written to the server. Filesystem/OS-specific compression is used, including skipping swap partitions; generic `zlib`-based [4, 21] compression is used on allocated blocks. Partitions that contain unknown filesystem types are either compressed generically or, optionally, entirely skipped.

Multicast: Our distribution mechanism uses a custom application-level receiver-initiated multicast protocol with NAK-avoidance [10]. In turn, this protocol relies on IP multicast support in the network switches to provide one-to-many delivery at the link level. The number of control messages is kept under control by multicasting client repair requests (NAKs), so that other clients can suppress duplicate requests. In terms of the multicast design space put forth in RFC 2887 [8], our application design requires only scalability and total reliability. We do not require other constraints, in particular ordered data or server knowledge of which clients have received data.

Two-level segmentation: We now address the issue of the granularity of data segments. Since we will need to resend lost multicast packets, yet do not need to preserve ordering, it is clear that we want the client-side decompression routines to process data segments out of order. Therefore, each data segment must be self-describing and stand alone as a decompressible unit. Since compression routines optimize their dictionaries based on the distribution of their input data, they achieve better compression ratios when given longer input to sample. That argues for longer segments.

Since Frisbee's basic job is I/O—copying disks through memory over networks—the classic hardware and OS architecture reasons that favor sequential I/O for its speed and efficiency also favor long segments. However, to preserve network and machine resources, we want our multicast loss recovery algorithm to use selective retransmission, which requires relatively short segments. Finally, we want a small segment size that fits into the Ethernet MTU.

The fundamental problem is that we need to follow the principle of Application Level Framing (ALF) [3], yet have conflicting application requirements. We address these conflicting demands by using a two-level seg-

mentation scheme. The unit of compression is the self-describing 1MB *chunk*, composed of 1024 1KB *blocks*. For the initial network transmission, the server multicasts an entire chunk, capping its rate by pausing every N (currently 16) blocks for a tunable period. Receivers selectively request missing blocks via partial request messages. In this way we achieve long segments that can efficiently be randomly-accessed, composed of small blocks that are typically, but not always, accessed sequentially. Subsets of blocks, as specified in receiver partial request messages, give us a flexible mechanism to request intermediate lengths, without undue complexity.

Specialization for resource availability: Since most modern Ethernet networks are switched, and the dedicated clients do not need bandwidth for other purposes, the main place that bandwidth must be saved is on the server and the server's network link, a situation for which multicast is ideally suited. The protocol is client-driven; this way, a server can be running at all times, but naturally falls idle when no clients are present. Additionally, this client-side control provides a high degree of robustness in the face of client failure and reduces server-side bookkeeping.

Receiver concurrency control: To install an image, we boot into a small, memory filesystem-based Unix system similar to the one used when creating the image. Using multiple threads, our disk loader client program takes care to overlap the computationally expensive decompression with the slow disk I/O. Using filesystem-specific compression turns out to give the biggest performance improvements at this stage—once compression is used to reduce the data transferred on the network, and maximal processor/I/O overlap is achieved, the bottleneck in performance becomes disk writes. We thus obtain a huge savings by not having to write unnecessary disk blocks. The ability to write free areas of the disk is still available because, as discussed in Section 4.4, this may be needed for confidentiality reasons.

Security: Since users must register to use Netbed, our threat model does not encompass determined malicious local adversaries. We have not yet needed to ensure security in the face of malicious users. We do expect to provide more security eventually, perhaps by signing image data or with VLAN technology. In Emulab, images are distributed via the "control" network, a single switched virtual LAN connecting all nodes. Thus there is the potential for an experiment to observe data on, or inject data into, a Frisbee multicast stream for another experiment. Our focus is on preventing accidental interference between experiments, in particular ensuring the integrity of image data. We use a simple check of the source IP address of incoming block data, which, although maliciously spoofable, suggests that it comes from the approved host. We do not currently provide an

option for ensuring privacy of distributed images. User-created images are protected via filesystem permissions while stored on the Frisbee server's disk.

4.2 Image Creation

In the Frisbee system, the *imagezip* application is responsible for creating images of either entire disks or single partitions. The images are compressed using both conventional and filesystem-aware techniques in a two-stage process. In the optional first phase, the partitions of interest are analyzed to determine if filesystem-aware compression can be done. Partitions are identified either explicitly by command line options or implicitly by reading the partition type field in the DOS partition table (on x86-based systems). Frisbee currently handles BSD FFS, Linux ext2fs and Windows NTFS filesystems as well as BSD and Linux swap partitions. If a partition is recognized, a filesystem-specific module is invoked to scan the filesystem free list and build up a list of free blocks in the partition. If a partition is not recognized, *imagezip* treats all blocks as allocated.

imagezip also has a limited ability to associate "relocation" information with data in a created image. This information allows it to create single partition images that can be loaded onto a disk that has a different partition layout. This facility is needed for filesystem types that contain absolute rather than partition-relative sector numbers. Notable examples of this are FreeBSD disklabels and LILO bootblocks.

In the second phase, the allocated blocks are read sequentially and compressed, producing 1MB chunks. Each chunk has a fixed-sized header with index information identifying the ranges of allocated blocks contained within it. Since the degree of compression is unpredictable, it is impossible to know exactly how much input data is required to fill the remaining space in a chunk. We counter with a simple algorithm that compresses smaller and smaller pieces as the chunk gets close to full; we then pad the chunk out to exactly 1MB. The padding typically runs around 20KB.

In summary, as shown in Figure 1, *imagezip* uses knowledge of filesystem types as well as conventional zlib compression to compress disk images. Images are segmented into self-describing 1MB chunks, each with independently compressed data.

4.3 Image Distribution

A compressed disk image in the Frisbee system is just a regular, albeit potentially very large, file and thus can be distributed in any number of ways, such as via *scp* or *NFS*, and then installed using the *imageunzip* command line program described in Section 4.4.

In a local area network environment, a more efficient and scalable way of image distribution is to use the Fris-

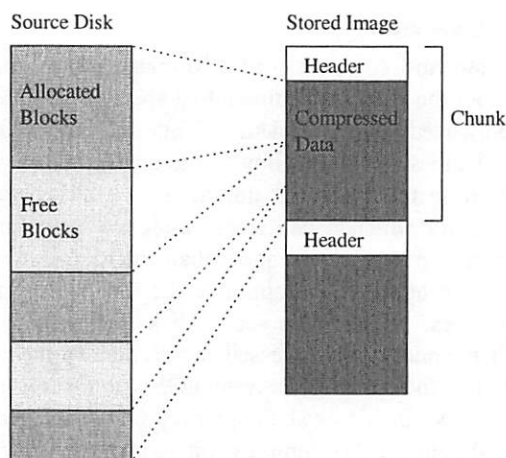


Figure 1: Image creation with *imagezip*

bee protocol as implemented in the *frisbeed* server and *frisbee* client. *Frisbeed* accepts request messages from multiple Frisbee clients and uses UDP on IP-multicast to transfer an image. Each *frisbee* client uses the multicast channel to request pieces of the desired image as needed until all pieces have been received, decompressed and written to the target disk. In the current implementation, each instance of *frisbeed* serves up a specific disk image using a unique multicast address. The information about what disk image and multicast address to use is communicated out-of-band to the server and clients. In Emulab's case, the client learns this from the central Netbed database.

4.3.1 The *frisbeed* Server

The *frisbeed* server has two threads, one which receives incoming requests and one which processes those requests and multicasts image data to the network. The server receive thread fields three types of messages from clients. JOIN and LEAVE messages bracket a client's participation in a multicast session. The server's response to a JOIN message includes the number of blocks in the image.

Clients issue data REQUEST messages containing a chunk number and a block range; typically they request the entire chunk. The server receive thread places block requests on a FIFO work queue, after first merging with any already queued request that overlaps the requested data range.

The *frisbeed* transmit thread loops, pulling requests from the work queue, reading the indicated data from the compressed image file, and multicasting it to the network in Frisbee BLOCK messages. BLOCK messages contain a single 1KB block of data along with identifying chunk and block numbers. Since a request allows for multiple blocks to be specified, a single request from the work queue may generate multiple BLOCK messages.

In our current production system, the server's network bandwidth consumption is controlled by placing a simple cap on the maximum bandwidth used. Two parameters are used to implement the cap: a burst size and a burst gap. The burst size is the number of BLOCK messages that can be transmitted consecutively without pausing, while the burst gap is the duration of that pause. Ideally, just an inter-packet delay could be used to pace data to the network, but the resolution of UNIX sleep mechanisms is dictated by the resolution of the scheduling clock, which is typically too coarse (1-10ms). Our current values of burst size (16) and gap (2ms) were empirically tuned for our environment. Clearly, this capping mechanism is adequate only on a dedicated server machine in a switched LAN environment, as the server does not adjust its transmission rate in response to network load. The effect of this is shown, and an alternative mechanism discussed, in Section 5.3.

4.3.2 The *frisbee* Client

The *frisbee* client is structured as three threads in order to overlap network I/O, disk I/O, and decompression. The network thread, whose basic operation is shown in Figure 2, is responsible for retrieving BLOCK messages multicast by the server, accumulating the contained data blocks into complete chunks, and queuing those chunks for processing by the decompression thread. The network thread also ensures that data arrives in a timely fashion by issuing REQUEST messages for needed chunks and blocks. The decompression thread dequeues completed chunks, decompresses the data and, using the index information from the chunk header, queues variable-sized disk write requests. The disk thread dequeues those requests and performs the actual disk write operations. Once all chunks have been written to disk, the client exits. The remainder of this section focuses on the acquisition of data via the Frisbee protocol.

A *frisbee* client will of course receive not only blocks it has explicitly requested, but those that other clients have requested as well. Ideally, *frisbee* would be able to save all such blocks. However, since blocks for a given chunk must be kept until the entire 1MB chunk has been received, and a compressed image may be hundreds to thousands of megabytes, this is not practical. Thus, *frisbee* maintains a cache of chunks for which it has received one or more blocks, discarding incoming data for other blocks when the cache is full. Currently, the size of this cache (typically 64MB) is configured via a command line parameter and is fixed for the duration of the client run.

The client keeps a timestamp for each outstanding chunk in the image, recording when it last issued a request for the chunk or observed another client's request for it. The timestamp prevents the client from re-requesting data too soon. Before a partial or full chunk

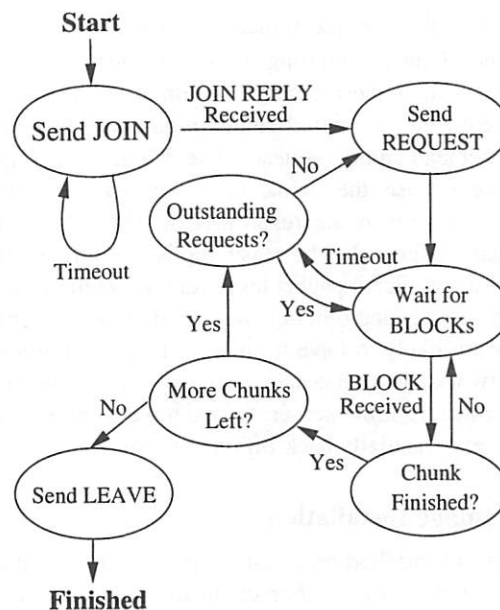


Figure 2: Basic operation of the *frisbee* client's network thread.

request is made, the client verifies that no client has requested the same chunk recently. *Frisbee* can track other clients' requests because all client-initiated messages (JOIN, LEAVE and REQUEST) are multicast.

After a client joins a session, it sends one or more REQUEST messages to start the transfer process. Instead of having each client request chunks in sequential order, clients randomize their initial request list. This prevents the clients from synchronizing, requesting the same chunks at the same time, which would cause *Frisbee*'s NAK-avoidance to perform less well. Each client is allowed to "request ahead" a fixed number of 1MB chunks.

Once a client has started and made its initial chunk requests, there are two situations in which it may make additional requests: when it has just completed a chunk and handed it off to the decompression thread or when it hasn't seen any packets (messages) for some period of time. The former represents the normal operation cycle: a client receives chunks, decompresses and writes them out, and makes further requests. When requesting new data following chunk completion, the first priority is completing any chunks for which some blocks have already been received. For each incomplete chunk currently in the client's cache, that chunk's timestamp is checked and, if it has been long enough since the chunk was last requested, the client issues a partial-chunk request to retrieve missing data for that chunk. Prioritizing partial-chunk requests over those for new chunks helps keep the decompression and disk threads busy and flushes data from the cache buffers sooner, making space available for new chunks. After handling partial-chunk requests, the client may also issue one or more full chunk

requests to fill its request-ahead window.

If the client is initiating a request due to a receiver timeout, the request process is similar to the chunk-completed case: partial-chunk requests followed by request-ahead chunk requests. The difference is that, in the timeout case, the chunk timestamp is not consulted; the requests are made regardless of when the chunks were last requested. The reasoning is that a timeout indicates a significant packet loss event between this client and the server (and other clients), so that even recent requests are likely to have been lost. To prevent flooding the network with requests in the event of a prolonged disconnection from the server, for example a server crash, clients exponentially back-off on requests.

4.4 Image Installation

Images are installed on a disk by one of two client programs. One is the *frisbee* client discussed above; the other is a simple program called *imageunzip*. They differ only in how they obtain the image: *imageunzip* reads an image out of a file while *frisbee* uses the Frisbee protocol to obtain it from the network. Both clients share the code used to decompress the data and write it out to disk. This section describes the operation of that common code.

Since the disk image is broken into independent 1MB chunks, the decompression code is invoked repeatedly, once for each chunk. For each chunk, the header is read to obtain ranges of allocated blocks contained in the chunk. For each allocated range, the indicated amount of data is decompressed from the chunk and queued for the disk writer thread to write to the appropriate location. The separation of decompression and disk I/O allows a great deal of overlap since raw disk I/O in FreeBSD is blocking. For free areas between ranges, the client can either skip them or fill them with zeros. The former is the default behavior and speeds the installation process dramatically in images with a large proportion of free space. However, this method may be inappropriate in some environments since it can “leak” information from the previous disk image to the current. For example, in Emulab where machines are time shared between experiments, some users may wish to have all their data “wiped” from their machines when they are done. For these environments, the installation client can be directed to zero-fill free space.

5 Evaluation

In this section, we evaluate the performance of our disk imaging and loading system, testing the speed of individual parts, as well as the entire system, with a variety of disk image properties, client counts and network conditions. Furthermore, we compare the performance of our

Image	FS Size	Data Size	Pct. Free	Comp. Type	Compressed Size	Time
Small	3067	624	79%	Naive	678	627
				FS-Aware	180	146
				Savings from FS-awareness		74% 77%
Large	3067	1776	42%	Naive	944	685
				FS-Aware	655	416
				Savings from FS-awareness		31% 39%
XP	4094	1894	64%	Naive	1688	968
				FS-Aware	575	282
				Savings from FS-awareness		66% 70%

Table 1: Performance of *imagezip* on three evaluation filesystems using both naive and filesystem-aware compression. Sizes are in (1024x1024) megabytes and times are in seconds.

system with a similar popular commercial offering and with a differential update program.

For our tests, we use one or more of a standard set of three test images. Our “small” image is a typical clean installation of FreeBSD on the FFS filesystem, which uses 642MB (21%) of a 3067MB filesystem. Our “large” image is a similar installation of FreeBSD that contains additional files typically found on a desktop workstation, such as several large source trees, compressed source archives, build trees, and additional binary packages; this image uses 1776MB (58%) of the available filesystem space. For comparison with Symantec Ghost, which performs best with NTFS filesystems, we used our “XP” image, which is a typical clean installation of Microsoft Windows XP Professional Edition. It uses 990MB of data with a 384MB swap file and 520MB “hibernation” file for a total of 1894MB (46%) of disk space in a 4094MB filesystem. All tests were performed on Emulab.¹

5.1 Image Creation with *imagezip*

To characterize the performance of reading and compressing disk image files, we ran *imagezip* on our large and small images. The output file was discarded, rather than written, to isolate the image creation time from time spent writing the created image to a remote filesystem or local disk. We used both filesystem-aware and naive compression. Results are shown in Table 1. As expected, the savings obtained by using filesystem-aware compression are roughly proportional to the amount of free space on the disk. Compression speed is more than adequate

¹In this evaluation, the clients are 850MHz Pentium IIIs, with an Intel 440BX motherboard chipset, 512MB RAM, and a 100MHz system bus. Their disks are 40GB IBM 60GXP 7200RPM IDE drives running at ATA/33, with 2MB buffers. The measured sustainable write speed to the region of the disks used in our tests is 21.4MB/second. The server is a 1.5GHz Pentium IV with 256 MB of PC133 RAM and an ATA/100 IDE disk. The clients are connected at 100Mbps and the server at 1000Mbps to a single switched LAN on a Cisco Catalyst 6509.

Target	Small FS, naive compress.	Small FS, FS-aware compress.	Large FS FS-aware compress.
null	98	21	65
disk	155	33	86
null (1 thread)	96	21	65
disk (1 thread)	242	50	145

Table 2: Time in seconds to decompress and install images from memory with both single- and multi-threaded *imageunzip*. The large naively compressed image was too large to fit into memory on our test nodes, and thus was not tested.

for our application, where images are usually generated once and used many times. Additional optimization is likely possible by multithreading the disk read and compression tasks, and eliminating internal data copies.

5.2 Image Installation with *imageunzip*

To characterize the performance of decompressing and writing disk images (independent of network distribution), we ran *imageunzip* on our large and small image files, reading the images from a memory-based filesystem. *imageunzip* uses the same decompression and disk writing code as the Frisbee client. For each test, the image file was read from the memory filesystem, decompressed, and written to disk. A second set of tests isolated decompression performance by discarding the decompressed image rather than writing it to disk. To measure the effectiveness of overlapping decompression and disk writing we repeated the tests, disabling multithreading in *imageunzip* so that a single thread both decompresses and writes the data. Table 2 contains the results.

By comparing the first two columns, we see significant savings from using filesystem-aware compression: the small image, with 80% free space, sees a time savings of 78% over the naively compressed image. From the last two rows, where there is only a single thread and thus no overlap of decompression and disk writing, we see that disk write speed is the limiting factor. Writing to disk accounts for 55–60% of the total time. Since disk writes are synchronous and the majority of the time is spent waiting, decompressing in parallel effectively hides much of its cost. This is demonstrated in the difference between the single- and multi-threaded results in which the multithreaded case is up to 40% faster.

5.3 Image Distribution with Frisbee

Scaling: To show Frisbee’s speed and scalability, we ran a number of tests, reloading sets of clients ranging in number from 1 to 80. During these tests, all clients began loading at the same time. Figure 3 shows the average client runtime for the small and large images using both naive and filesystem-aware compression. The minimum and maximum times are indicated with error

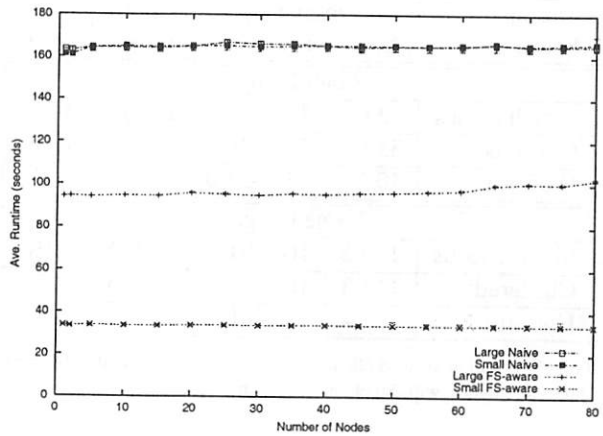


Figure 3: Frisbee client scaling from 1 to 80 nodes.

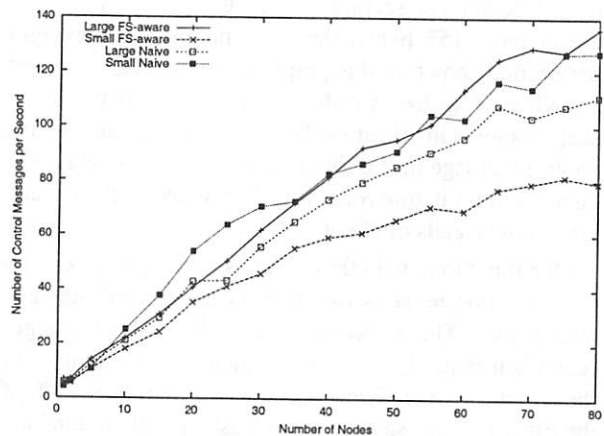


Figure 4: Average number of messages per second received by the *friseed* server for the scaling experiment in Figure 3.

bars, but the variance is so low that they are not identifiable at the default magnification. Frisbee is fast and scalable: it loads the small image onto one node in 33.8 seconds, and onto 80 nodes in 33.6 seconds. It loads the naively-compressed images in nearly constant time. For the filesystem-aware large image, the runtime does increase slowly: Frisbee loads 1 node in 94 seconds and 80 nodes in 102 seconds. The reason for this difference remains to be explored; we suspect that the fraction of partial requests may be increasing, or the clients’ chunk buffers may be filling up.

Across all runs, Frisbee’s network efficiency is very good; the number of duplicate blocks transmitted due to packet loss or duplicate requests did not exceed 8% of the total blocks sent. Note the nearly identical runtimes for the two naively compressed images, despite the nearly 50% difference in their compressed sizes. Since both must write a full 3GB of data to disk, this demonstrates that the disk is indeed the bottleneck on these machines.

Startup Scenario	Runtime (s)		Client msgs	Dup Data
	Ave	Range		
Small Image				
Simultaneous	33.6	32.9–34.7	2753	3.2%
Clustered	35.6	33.2–40.3	4561	46%
Uniform	40.0	34.5–51.0	7875	59%
Large Image				
Simultaneous	100.2	100–101	12772	7.3%
Clustered	113.3	106–126	17266	26%
Uniform	132.4	120–147	23842	37%

Table 3: Effect of skewed client start times on Frisbee load of the small and large images, with 80 clients under three scenarios.

Figure 4 shows the average number of control messages (JOIN, REQUEST, and LEAVE messages) received by the server per second. Since the control messages are at most 152 bytes, the peak number of messages per second shown in this graph, 127, represents at most 154Kbps of upstream traffic to the server. If the linear scaling shown in this graph holds for larger client counts, control message traffic should not run into packet rate or bandwidth limitations on a 100Mbps LAN until we reach tens of thousands of clients.

One thing to note in these graphs is that the maximum node runtime remains flat even as the control message traffic rises. This is because the *frisbee* server merges redundant requests in its work queue. For example, in the worst case at 127 messages per second, over 93% of the REQUEST messages were at least partially redundant. This indicates that there is considerable opportunity for improvement in the NAK avoidance strategy, a topic discussed later.

Another important result is that load times with Frisbee are very similar to the load times reported in Table 2 for *imageunzip*: Frisbee is able to keep the disk-writing thread supplied with data at a high enough rate that network transfer rate is not the bottleneck. With respect to supplying the disk writer, Frisbee's multicast distribution provides nearly the same level of performance as reading from local RAM on the client, and maintains this performance for a large number of clients.

Skewed Starting Times: We examined Frisbee's performance when client nodes are not started simultaneously. In practice, this can occur when clients are not rebooted simultaneously, when their boot durations vary, or when they are rebooted in groups. In this test, shown in Table 3, we loaded the small and large images on 80 clients under three different scenarios. In the first, all 80 clients start loading simultaneously, as in the scaling tests of the previous section. This is the idealized Emulab large experiment creation situation. In the second, clients are started in four groups of 20 at 10 second intervals.

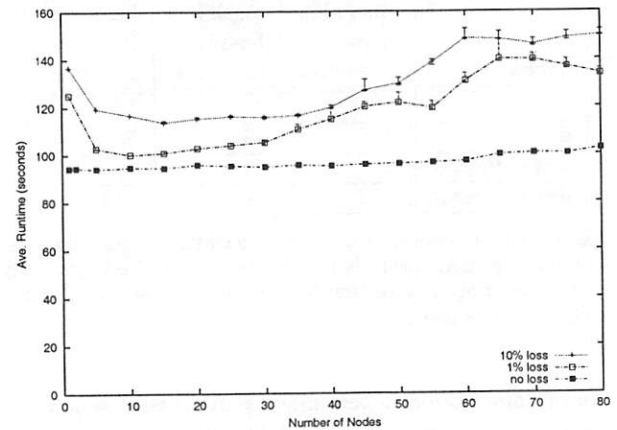


Figure 5: Frisbee client scaling from 1 to 80 nodes with packet loss. Error bars show the minimum and maximum times.

This is a realistic representation of Emulab's current behavior, where node reloads are effectively clustered by staggering reboots in groups, to avoid boot time scaling problems related to PXE, DHCP and TFTP. Finally, we uniformly distribute start times of the 80 clients over a 30 second interval, the same interval over which the clients were started in the cluster test.

As one might expect, skewing requests results in redundant block transfers. Late joining clients miss the blocks requested by earlier clients and thus request them again. This, in turn, stalls the early joining clients when the server sends redundant data. As a result, late joining clients tend to finish significantly faster. The Frisbee client could be made more fair by making its request behavior less aggressive. Currently, the client issues its own requests even if it is constantly receiving sufficient data to keep it busy. Making the client more passive, requesting data only when not making forward progress, would restore fairness. That change risks causing a client's runtime to further increase, if it doesn't quickly enough transition to making its own requests, and falls idle. However, even in the current state, we consider the ability to introduce clients into a Frisbee run at any time to be well worth the increases in client runtime and resource consumption.

Packet Loss: In general, we do not expect Frisbee to have to contend with packet loss, since its target environment is a switched LAN in which the receivers are dedicated clients. However, packet loss can still occur if the server or switch is overloaded. To investigate Frisbee's behavior in the presence of packet loss, we loaded the large image on 1 to 80 clients with packet loss rates of 0%, 1% and 10%. Packet drops were done at the server; since Frisbee clients are running only Frisbee, there will be no contention for their links. Figure 5 summarizes the results. Packet loss makes Frisbee more sensitive to the

# of Clients		Ave. Runtime (s)		Client	Dup
pc600	pc2000	pc600	pc2000	msgs	Data
Server at 70Mbit/sec					
0	4	—	94.3	895	0.0%
1	3	96.7	94.0	918	0.6%
3	1	95.9	93.9	885	0.2%
4	0	95.5	—	877	0.2%
Server at 90Mbit/sec					
0	4	—	72.3	667	0.1%
1	3	105.0	81.6	986	24.0%
3	1	106.7	93.7	1222	30.8%
4	0	106.5	—	1186	28.7%

Table 4: Effect of combinations of heterogeneous clients on Frisbee load of large image with two different server bandwidths.

number of clients, and there is definitely room for improvement, since with a large number of nodes, the 1% packet loss case performs similarly to the 10% case. Still, performance is clearly acceptable for what we expect to be a rare occurrence. It is interesting to note that a single client performs worse with loss than do multiple clients. When there is a single client, and a REQUEST message it sends to the server is lost, a timeout must pass before it will ask again. During that time, the client is idle. When there are multiple clients, blocks sent as the result of their requests will enable the first client to make progress until its timeout period expires.

Heterogeneous Clients: Thus far we have run all tests on clients of the same type. This reflects the current node base of Emulab, in which the majority of nodes are of the same type. However, given the pace of technology, it is typical for a large collection of machines gathered over time to be much more diverse. To gain a feel for how Frisbee would perform in such an environment, we performed a small-scale experiment using combinations of machines of two widely different types loading the large image. A *pc600* is a 600MHz processor with 100MHz SDRAM and an ATA/33 hard drive while a *pc2000* is a 2GHz processor with 400MHz RDRAM and an ATA/100 hard drive. Both have 100Mbit ethernet interfaces. The large difference in CPU and memory speed enables the *pc2000* to decompress data much faster. The higher frequency disk interface, coupled with a newer-generation hard drive, also allows it to write much faster (38.8 MB/sec vs. 21.4 MB/sec). The hypothesis is that the *pc2000*s will request blocks at a much higher rate than the *pc600*s, causing the latter to miss blocks and make many more re-requests. These re-requests will in turn slow the effective data rate to the *pc2000*s. Results are shown in Table 4.

The top half of the table shows runs using the default server network bandwidth of 70Mbps, a value tuned to

efficiently support the 600-850Mhz class of machines in Emulab. Here we see that runtimes are very similar for all combinations. However, the lack of improvement by the *pc2000*s is because they are throttled by the network bandwidth, not by the presence of slower machines. This is illustrated in the lower half of the table, where the server bandwidth was increased to 90Mbps. At this rate, a set of four *pc2000*s is able to load the image 23% faster, while a set of four *pc600*s takes 12% longer. In this configuration, we do see an effect when combining the two types. Combining a single *pc600* with three *pc2000*s slows the faster machines, increasing their runtime to 81.6 seconds, while the slower machine runtime remains unchanged. With three *pc600*s and a single *pc2000*, the latter is further slowed to 93.7 seconds, with little change for the *pc600*s. This slowdown is directly attributable to the increase in duplicate data caused by the slower machines' re-request messages. While not shown in the table, the duplicate data rate tops out at 35% with eight *pc600*s. At this rate, the *pc2000* continues to run faster than the *pc600*s, taking 102 seconds versus 112 for the the slower machines.

NAK Avoidance: We ran Frisbee with its NAK avoidance features, snooping on control messages and time-limiting of re-requests, disabled. With 80 clients, the message received rate at the server increased dramatically, from 85 per second to 264 for the small image, and from 146 per second to 639 for the large image.

As noted earlier, the NAK avoidance features still seem to allow a large number of spurious control messages, which are then ignored by the server. These messages are the result of using a static time limit (one second) for re-requests. When the limit is changed to two seconds, the request rate is reduced to 47 per second for the small image and 84 for the large image. However, blindly increasing the static value can result in increased client runtime when small numbers of nodes are involved and messages are truly lost. Ideally, we need to take into account the transfer rate of the server and the length of the server's work queue (which varies with the number of active clients), both of which affect the latency of an individual request. A dynamic time limit could be implemented by having the server piggyback current bandwidth and queue length information on BLOCK messages. Clients would use that information to calculate a more appropriate re-request rate.

Server Load: Although we have demonstrated that the Frisbee client performs well in a variety of situations, another important consideration is how the *frisbeed* server performs. In this section we consider the CPU, disk and network resources required for a single server instance, as well as for multiple instances running on the same host.

As *frisbeed* essentially just moves data from the disk

Startup Scenario	Server Runtime	Client msgs	Data xfer Rate (MB/s)	CPU use (%)
Small Image				
At once	34.7	2753	5.36	9.9
Clustered	55.1	4561	6.02	12.0
Uniform	65.7	7875	6.67	12.3
Large Image				
At once	101.0	12772	6.99	14.5
Clustered	126.5	17266	7.05	14.5
Uniform	150.1	23842	6.95	14.2

Table 5: Server load observed during skewed client startup tests.

Servers x Clients	Ave. Srv. Runtime (s)	Data xfer rate (MB/s)	Total CPU use (%)
1 x 80	34.7	5.36	9.9
2 x 40	35.2	11.3	32.0
4 x 20	56.5	23.0	51.6
8 x 10	58.1	31.3	72.0

Table 6: Server load with multiple, concurrent *frisbeed* servers loading the small image. The CPU time used by CPU and network monitors is not included—at 8 servers, the CPU is saturated.

to the network, we would expect the use of all three resources to increase with the number of BLOCK messages processed. As seen in the client performance measurements, increased requests most commonly occur when client startup is skewed or there is significant packet loss, causing the server to resend data. Table 5 details the run time, CPU use and amount of data transferred from disk to network for the skewed client experiment reported in Table 3. The rate of CPU and disk use is bounded by the network send rate which, as mentioned in Section 4.3.1, is controlled by a simple static bandwidth cap. The value of 70Mbits/sec used in our evaluation, which includes all network overhead, translates to 7.7MB per second of image data. In the table we can see that as the data transfer rate approaches this value, CPU use does not exceed 15%.

More problematic is the multiple server scenario. With no provision for dynamically altering bandwidth consumption, resource use is additive in the number of running servers. Table 6 demonstrates this effect as we run from one to eight *Friseed* instances to load 80 client nodes. Even with a 1000Mbps link from the server, at two *Friseeds* we are near the 100Mbps limit of the client links and the switch begins to drop packets. By eight *Friseeds*, the CPU is saturated. Moreover, not shown in this table is the lack of fairness between servers. For example, in the four-server case one finished in 35 seconds while the other three took longer than 60 seconds.

While we can tolerate this behavior in the current Emulab, where server and switch resources are plentiful,

a better solution is needed. Recently we have prototyped a rate-based pacing mechanism so that *frisbeed* will adapt to network load. We use a simple additive-increase multiplicative-decrease algorithm which dynamically adjusts the burst size based on the number of lost blocks. The key to calculating the latter is that the server can treat any partial chunk request as indicating a lost packet. Results for this version of *Friseed* are mixed, with two servers quickly adapting to each take half the 100Mbps bandwidth, but with four and eight servers wildly oscillating. We believe the latter is merely a consequence of our simplistic rate-equation and not a reflection on the Frisbee protocol and its ability to detect loss events.

5.4 Comparison to rsync

To get some idea of the speed of our disk imaging approach compared to differential file-based approaches, we ran *rsync* on the three filesystems in our small image, with essentially no changes between source and target machines. We configured *rsync* to identify changed files but not to update any. This is a best-case test for *rsync*, since its runtime strongly depends on the amount of data it must copy.

We found that, when identifying changed files based solely on timestamps, *rsync* is approximately three times faster than Frisbee—it took 12 seconds to compare two machines vs. Frisbee's 34 seconds to blindly write the same image. Security and robustness concerns, however, prevent us from using timestamps as an accurate way of comparing files, since they are not reliable when experimenters have full root access. When *rsync* performs MD4 hashes on all files to find differences, its runtime increases to 170 seconds, five times longer than Frisbee. Given our static disk distribution needs, some domain-specific optimizations to *rsync* should be possible. For example, while it must always hash the target disk's files, the server can cache the hashes of its unchanging source disk. In the above test, server-side hashing accounted for approximately 60 seconds of *rsync*'s runtime and is serialized with client-side processing. Therefore, this optimization should reduce *rsync*'s runtime to three times Frisbee's.

However, these small tests still demonstrate that, on a fast distribution network where bandwidth is not the major bottleneck, and with disk contents such as ours, it is unnecessary to spend time identifying changed files. It is faster simply to copy the entire disk.

5.5 Comparison to Ghost

We compared Frisbee to one of the most popular commercial disk imaging packages, Symantec Ghost. Ghost has a similar feature set, including filesystem-specific compression and multicast distribution. Ghost's "high"

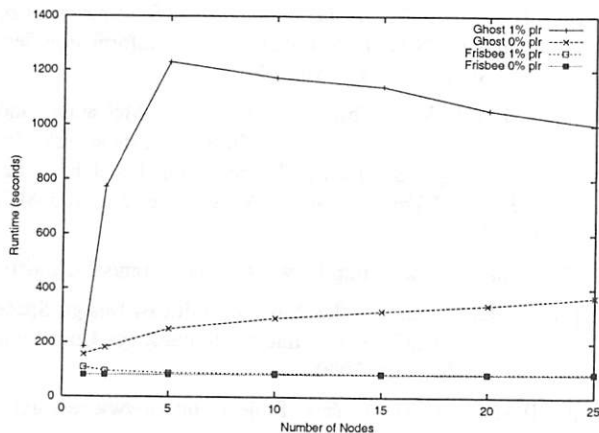


Figure 6: Scaling of Frisbee and Ghost with and without 1% packet loss.

compression setting (level four out of nine) appears substantially similar to *imagezip*'s compression (using *zlib* level four). We used the Windows XP disk image for this comparison, which *imagezip* compressed to 575MB and Ghost compressed to 594MB. Both Ghost and Frisbee have the ability to skip the swap and hibernation files, whose contents do not need to be preserved.

Figure 6 shows Frisbee and Ghost load times on 1 to 25 clients, with no packet loss and with a 1% loss rate. Since Ghost is a commercial product with per-client licensing, the maximum number of clients we tested was limited by licensing costs. Still, clear trends are visible: Ghost's base (one-client) time of 156 seconds is nearly twice as high as Frisbee's 81 seconds, and it increases with the number of clients to 369 seconds, while Frisbee's grows only 5% to 85 seconds. Frisbee exhibits excellent tolerance to 1% packet loss. The extremely poor behavior of Ghost in the presence of packet loss is remarkable, and bears further investigation.

An important difference between Frisbee and Ghost is that Frisbee allows new clients to connect while other clients are in the process of receiving an image. Ghost, on the other hand, requires all clients to start simultaneously. This substantially impacts the latency of the system, as all clients must wait for the slowest to begin, and clients that wish to join after a session has been started must first wait for the ongoing session to finish. One can work around this restriction by starting a new Ghost session for the same image, with the downside of unnecessarily increasing network traffic.

6 Related Work

Partition Image [16] is an open-source program for creating and restoring disk partition images. Like Frisbee, it uses filesystem-aware compression in conjunction with conventional compression to reduce the size of the image

and accelerate image distribution and installation. Partition Image currently supports a larger set of recognized filesystem types. Unlike Frisbee, images are compressed as a single unit and thus the image must be decompressed sequentially. Partition Image also does not support creating complete disk images with multiple partitions. Partition Image uses a stream-oriented unicast protocol with optional encryption. Thus it will not scale as well as Frisbee's multicast protocol, but will work unchanged in a wide-area network environment. The Partition Image client can both save and restore images over the network where Frisbee currently has no built-in mechanism for saving images across the network.

HCP [18] is a hybrid technique for synchronizing disks, using a form of differential updating, but below the file level. HCP is one method used in Stanford's Collective project to copy virtual disks ("capsules") between machines. In HCP, a cryptographic hash is used to identify blocks in the client and server disks. To synchronize a block between the two, the client first requests the hash for the desired block and compares that to the hash for all blocks in all local virtual disks. If any local block matches the hash, that block is used to provide the data, otherwise the actual block data is obtained from the server. HCP takes advantage of the high degree of similarity between the multiple virtual disks that could reside on any client and the fact that the same virtual disk will tend to migrate back and forth between a small set of machines. Still, as noted by the authors, HCP is only appropriate in environments where the network is the bottleneck due to increased disk seek activity on the client.

Numerous other multicast protocols for bulk data transfer have been proposed, such as SRM [6] and RMTP [11]. Frisbee's target environment, high-speed, low packet loss, low-latency LANs, allows a much simpler protocol, which can be optimized for very high throughput. In the taxonomy of known multicast protocols presented in [10], the Frisbee protocol is considered a RINA (Receiver Initiated with NAK-Avoidance) protocol.

7 Future Work

Extending the Frisbee system from a LAN environment into the wide area presents an interesting challenge. In addition to its Emulab cluster, Netbed manages a number of nodes at sites around the world. Currently, images compressed by *imagezip* are distributed via unicast, and installed with *imageunzip*, but this will clearly not scale for a large number of nodes or frequent image distribution. Extending diskloading to the wide area will assuredly raise issues that are not present in our LAN environment. Some of these issues, such as differing client bandwidths and TCP-friendliness, have been the subjects

of extensive research and we will undoubtedly be able to leverage this work. For example, techniques such as those employed by Digital Fountain [1] or WEBRC [12] may be useful. Digital Fountain uses a multicast protocol based on erasure codes [13] to create a large-scale software distribution system. WEBRC obtains an estimate of multicast RTT for flow control and TCP friendliness, and uses multiple multicast streams and a fluid model to serve clients of differing bandwidths.

When sending data in the wide area, security is also a concern—while it is acceptable to send images unencrypted and unauthenticated on a tightly-controlled LAN, care will have to be taken in the wide area to ensure that eavesdroppers cannot obtain a copy of sensitive data on the image, or alter disk contents.

8 Conclusion

We have presented Frisbee, a fast and scalable system for disk image generation, distribution in local area networks, and installation. We summarized our target application domain and have shown how aspects of that domain governed our choices in designing the system. As well as discussing our use of established techniques, we have explained our methods of filesystem-aware compression and two-level segmentation, and how they are particularly well-suited to our multicast file transfer protocol. Finally, we have shown that this system exceeds our performance requirements and scales remarkably well to a large number of clients.

Acknowledgments

Many thanks to Kirk Webb for gathering important performance results, to Russ Christensen for implementing NTFS compression, to Dave Andersen for implementing an early unicast disk imager in the OSKit, and to the anonymous reviewers for their useful feedback.

References

- [1] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *Proc. of ACM SIGCOMM '98*, pages 56–67, Vancouver, BC, 1998.
- [2] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast Security: A Taxonomy and Some Efficient Constructions. In *Proc. of INFOCOM '99*, pages 708–716, Mar. 1999.
- [3] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proc. of ACM SIGCOMM '90*, pages 200–208, Sept. 1990.
- [4] P. L. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification version 3. Internet Request for Comments 1950, IETF, May 1996.
- [5] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, Apr. 1995.
- [6] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6):783–803, Dec. 1997.
- [7] Symantec Ghost. <http://www.symantec.com/sabu/ghost/>.
- [8] M. Handley et al. The Reliable Multicast Design Space for Bulk Data Transfer. Internet Request For Comments 2887, IETF, Aug. 2000.
- [9] IBM Corp. The Océano Project. <http://www.research.ibm.com/oceanoproject/>.
- [10] B. N. Levine and J. Garia-Luna-Aceves. A Comparison of Known Classes of Reliable Multicast Protocols. In *Proc. of IEEE ICNP '96*, pages 112–123, Oct. 1996.
- [11] J. C. Lin and S. Paul. RMTP: A Reliable Multicast Transport Protocol. In *Proc. of INFOCOM '96*, pages 1414–1424, San Francisco, CA, Mar. 1996.
- [12] M. Luby, V. K. Goyal, S. Skaria, and G. B. Horn. Wave and Equation Based Rate Control Using Multicast Round Trip Time. In *Proc. of ACM SIGCOMM '02*, pages 191–204, Aug. 2002.
- [13] A. J. McAuley. Reliable Broadband Communication Using a Burst Erasure Correcting Code. In *Proc. of ACM SIGCOMM '90*, pages 297–306, Philadelphia, PA, Sept. 1990.
- [14] J. Moore and J. Chase. Cluster On Demand. Technical Report CS-2002-07, Duke University, Dept. of Computer Science, May 2002.
- [15] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-bandwidth Network File System. In *Proc. of 18th ACM SOSP*, pages 174–187, Banff, AB, Canada, Oct. 2001.
- [16] Partition Image for Linux. <http://www.partimage.org/>.
- [17] rsync. <http://rsync.samba.org/>.
- [18] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proc. of OSDI '02*, pages 377–390, Boston, MA, Dec. 2002.
- [19] D. Towsley, J. Kurose, and S. Pingali. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. *IEEE Journal on Selected Areas in Communications*, 13(3), April 1997.
- [20] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of OSDI '02*, pages 255–270, Boston, MA, Dec. 2002.
- [21] zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library. <http://www.gzip.org/zlib/>.

Robust, Portable I/O Scheduling with the Disk Mimic

Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin, Madison

Abstract

We propose a new approach for I/O scheduling that performs on-line simulation of the underlying disk. When simulation is integrated within a system, three key challenges must be addressed: first, the simulator must be portable across the full range of devices; second, all configuration must be automatic; third, the computation and memory overheads must be low. Our simulator, the Disk Mimic, achieves these goals by building a table-based model of the disk as it observes the times for previous requests. We show that a shortest-mimicked-time-first (SMTF) scheduler performs nearly as well as an approach with perfect knowledge of the underlying device and that it is superior to traditional scheduling algorithms such as C-LOOK and SSTF; our results hold as the seek and rotational characteristics of the disk are varied.

1 Introduction

High-performance disk schedulers explored in the research literature are becoming progressively more tuned to the performance characteristics of the underlying disks. Each generation of disk schedulers has accounted for more of the behavior of storage devices at the time. For example, disk schedulers analyzed in the 1970s and 1980s focused on minimizing seek time, given that seek time was often an order of magnitude greater than the expected rotational delay [10, 26, 29]. In the early 1990s, the focus of disk schedulers shifted to take rotational delay into account, as rotational delays and seek costs became more balanced [13, 21, 31].

At the next level of sophistication, a disk scheduler takes all aspects of the underlying disk into account: track and cylinder switch costs, cache replacement policies, mappings from logical block number to physical block number, and zero-latency writes. For example, Worthington *et al.* demonstrate that algorithms that effectively utilize a prefetching disk cache perform better than those that do not [31].

However, the more intricate the knowledge a scheduler has of the disk, the more barriers there are to its realization within operating system kernels. Specifically, there are three obstacles that must be overcome. First, the scheduler must discover detailed knowledge of the underlying disk. Although a variety of tools have been described that auto-

matically acquire portions of this knowledge [19, 25, 32], it must still be embedded into the disk model employed by the scheduler; the resulting scheduler is then configured to handle only a single disk with those specific characteristics. Second, the disk scheduler must also have knowledge of the current state of the disk, such as the exact position of the disk head. Given that head position is not exposed by current disk controllers and its position is not predictable due to low-level disk techniques such as wear leveling, predictive failure analysis, and log updates, the scheduler must control the current position using non-trivial techniques [11, 33]. Finally, the computational costs of detailed modeling can be quite high [31]; it is not uncommon for the time to model request time to be larger than the time to service the request [4].

Due to these difficulties, few disk schedulers that leverage more than basic seek costs have been implemented for real disks. When considering rotational position, most previous work has been performed within simulation environments [13, 21, 23, 31]. The schedulers that have recently been implemented by researchers have either contained substantial simplifications [12] or have been painstakingly tuned for a small group of disks [11, 33]. Not surprisingly, the disk schedulers found in modern operating systems such as Linux, NetBSD, and Solaris, attempt only to minimize seek time.

1.1 A Different Approach

We believe that a promising alternative approach to embedding detailed knowledge of the disk into the scheduler is to embed an *on-line simulator* of the disk into the scheduler. An I/O scheduler is able to use on-line simulation of the underlying storage device to predict which request in its queue will have the shortest positioning time. Although a variety of disk simulators exist [4, 14, 30], most are targeted for performing traditional, off-line simulations, and unfortunately, the infrastructure for performing on-line simulation is fundamentally different.

In many respects, the requirements of an on-line simulator are more stringent than those of an off-line simulator. First, the on-line simulator must be *portable*; that is, the simulator must be able to model the behavior of any disk drive that could be used in practice. Second, the on-line simulator must have *automatic run-time configuration*, since one cannot know the precise characteristics

of the underlying device when constructing the simulator; it is highly undesirable if a human administrator must interact with the simulator. Finally, the on-line simulator must have *low overhead*; the computation and memory overheads of an on-line simulator must be minimized such that the simulator does not adversely impact system performance.

In addition to the complexity it introduces, an on-line simulator also provides ample opportunities for simplification. First, the on-line simulator has the opportunity to observe the run-time behavior of the device; not only does this allow the simulator to configure itself on the fly, it also allows the simulator to adjust to changes in the behavior of the device over time. Second, the on-line simulator can be specialized for the problem domain in question. Finally, the on-line simulator does not need to be parameterizable; that is, since an on-line simulator is not exploring different versions of the device itself, the simulator does not need to contain a functional model of the device.

1.2 Contributions

We address how to implement an I/O scheduler that is aware of the underlying disk technology in a simple, portable, and robust manner. To achieve this goal, we introduce the Disk Mimic, which meets the requirements of an on-line simulator for disk scheduling. The Disk Mimic is based upon a simple table-based approach, in which input parameters to the simulated device are used to index into a table; the corresponding entry in the table gives the predicted output for the device. A table-based approach is appropriate for on-line simulation because it can portably capture the behavior of a variety of devices, requires no manual configuration, and can be performed with little computational overhead. However, there is a significant challenge as well: to keep the size of the table tractable, one must identify the input parameters that significantly impact the desired outputs. The method for reducing this input space depends largely upon the domain in which the on-line simulator is deployed.

We show that for disk scheduling, two input parameters are sufficient for predicting the positioning time: the logical distance between two requests and the request type. However, when using inter-request distance for prediction, two issues must be resolved. First, inter-request distance is a fairly coarse predictor of positioning time; as a result, there is high variability in the times for different requests with the same distance. The implication is that the Disk Mimic must observe many instances for a given distance and use an appropriate summary metric for the distribution; experimentally, we have found that summarizing a small number of samples with the mean works well. Second, given the large number of possible inter-request distances on a modern disk drive, the Disk Mimic cannot record all distances in a table of a reason-

able size. We show that simple linear interpolation can be used to represent ranges of missing distances, as long as some number of the interpolations within each range are checked against measured values.

We propose a new disk scheduling algorithm, shortest-mimicked-time-first (SMTF), which picks the request that is predicted by the Disk Mimic to have the shortest positioning time. We demonstrate that the SMTF scheduler can utilize the Disk Mimic in two different ways; specifically, the Disk Mimic can either be configured off-line or on-line, and both approaches can be performed automatically. When the Disk Mimic is configured off-line, it performs a series of probes to the disk with different inter-request distances and records the resulting times; in this scenario, the Disk Mimic has complete control over which inter-request distances are observed and which are interpolated. When the Disk Mimic is configured on-line, it records the requests sent by the running workload and their resulting times. Note that regardless of whether the Disk Mimic is configured off-line or on-line, the simulation itself is always performed on-line, within an active system.

We show that the Disk Mimic can be used to significantly improve the throughput of disks with high utilization. Specifically, for a variety of simulated and real disks, C-LOOK and SSTF perform between 10% and 50% slower than SMTF. Further, we demonstrate that the Disk Mimic can be successfully configured on-line; we show that while the Disk Mimic learns about the storage device, SMTF performs no worse than a base scheduling algorithm (*e.g.*, C-LOOK or SSTF) and quickly performs close to the off-line configuration (*i.e.*, after approximately 750,000 requests).

The rest of the paper is organized as follows. In Section 2 we describe the SMTF scheduler in more detail and in Section 3 we describe the Disk Mimic. We describe our basic methodology for evaluation in Section 4. Next, we investigate the issues of configuring the Disk Mimic off-line in Section 5. We then describe the additional complexities of configuring the Disk Mimic on-line and show its performance in Section 6. Finally, we describe related work in Section 7 and conclude in Section 8.

2 I/O Scheduler

Many modern disks implement scheduling in the device itself. While this might suggest that file system I/O scheduling is obsolete, there are several reasons why the file system should perform scheduling. First, disks are usually able to schedule only a limited number of simultaneous requests since they have more restrictive space and computational power constraints. Second, there are instances when increased functionality requires the scheduling to be done at file system level. For example, Iyer and Druschel introduce short waiting times in the scheduler

to preserve the continuity of a stream of requests from a single process rather than interleaving streams from different processes [12]. Further, Shenoy and Vin implement different service requirements for applications by implementing a scheduling framework in the file system [23].

We now briefly describe the approach of a new file system I/O scheduler that leverages the Disk Mimic. We refer to the algorithm implemented by this scheduler as shortest-mimicked-time-first, or SMTF. The basic function that SMTF performs is to order the queue of requests such that the request with the shortest positioning time, as determined by the Disk Mimic, is scheduled next. However, given this basic role, there are different optimizations that can be made. The assumptions that we use for this paper are as follows.

First, we assume that the goal of the I/O scheduler is to optimize the *throughput* of the storage system. We do not consider the fairness of the scheduler. We believe that the known techniques for achieving fairness (*e.g.*, weighting each request by its age [13, 21]) can be added to SMTF as well.

Second, we assume that the I/O scheduler is operating in an environment with heavy disk traffic. Given that the queues at the disk may contain hundreds or even thousands of requests [13, 21], the computational complexity of the scheduling algorithm is an important issue [2]. Given these large queue lengths, it is not feasible to perform an optimal scheduling decision that considers all possible combinations of requests. Therefore, we consider a greedy approach, in which only the time for the next request is minimized [13].

To evaluate the performance of SMTF, we compare to the algorithms most often used in practice: first-come-first-served (FCFS), shortest-serve-time-first (SSTF), and C-LOOK. FCFS simply schedules requests in the order they were issued. SSTF selects the request that has the smallest difference from the last logical block number (LBN) accessed on disk. C-LOOK is a variation of SSTF where requests are still serviced according to their LBN proximity to the last request serviced, but the scheduler picks requests only in ascending LBN order. When there are no more such requests to be serviced, the algorithm picks the request in the queue with the lowest LBN and then continues to service requests in ascending order.

To compare our performance to the best possible case, we have also implemented a best-case-greedy scheduler for our simulated disks; this best-case scheduler knows exactly how long each request will take on the simulated disk and greedily picks the request with the shortest positioning time next. We refer to this scheduler as the greedy-optimal scheduler.

3 The Disk Mimic

The Disk Mimic is able to capture the behavior of a disk drive in a portable, robust, and efficient manner. To predict the performance of a disk, the Disk Mimic uses a simple table, indexed by the relevant input parameters to the disk. Thus, the Disk Mimic does not attempt to simulate the mechanisms or components internal to the disk; instead, it simply reproduces the output as a function of the inputs it has observed.

3.1 Reducing Input Parameters

Given that the Disk Mimic uses a table-driven approach to predict the time for a request as a function of the observable inputs, the fundamental issue is reducing the number of inputs to the table to a tractable number. If the I/O device is treated as a true black box, in which one knows nothing about the internal behavior of the device, then the Disk Mimic must assume that the service time for each request is a function of all previous requests. Given that each request is defined by many parameters (*i.e.*, whether it is a read or a write, its block number, its size, the time of the request, and even its data value), this leads to a prohibitively large number of input parameters as indices to the table.

Therefore, the only tractable approach is to make assumptions about the behavior of the I/O device for the problem domain of interest [3]. Given that our goal is for the I/O scheduler to be portable across the realistic range of disk drives, and not to necessarily work on any hypothetical storage device, we can use high-level assumptions of how disks behave to eliminate a significant number of input parameters; however, the Disk Mimic will make as few assumptions as possible.

Our current implementation of the Disk Mimic predicts the time for a request from two input parameters: the *request type* and the *inter-request distance*. We define inter-request distance as the logical distance from the first block of the current request to the last block of the previous request. The conclusion that request type and inter-request distance are key parameters agrees with that of previous researchers [18, 27].

We now briefly argue why inter-request distance and request type are suitable parameters in our domain. We begin by summarizing the characteristics of modern disk drives. Much of this discussion is taken from the classic paper by Ruemmler and Wilkes [18]; the interested reader is referred to their paper for more details.

3.1.1 Background

A disk drive contains one or more *platters*, where each platter *surface* has an associated disk head for reading and writing. Each surface has data stored in a series of concentric circles, or *tracks*. A single stack of tracks at a common distance from the spindle is called a *cylinder*. Modern disks also contain RAM to perform caching; the

caching algorithm is one of the most difficult aspects of the disk to capture and model [24, 32].

Accessing a block of data requires moving the disk head over the desired block. The time for this has two dominant components. The first component is *seek time*, moving the disk head over the desired track. The seek time for reads is likely to be less than that for writes, since reads can be performed more aggressively. A read can be performed when a block is not yet quite available because the read can be repeated if it was performed from the wrong sector; however, a write must first verify that it is at the right sector to avoid overwriting other data. The second component is *rotation latency*, waiting for the desired block to rotate under the disk head. The time for the platter to rotate is roughly constant, but it may vary around 0.5 to 1% of the nominal rate; as a result, it is difficult to predict the location of the disk head after the disk has been idle for many revolutions. Besides these two important positioning components there are other mechanical movements that need to be accounted for: head and track switch time. A head switch is the time it takes for the mechanisms in the disk to activate a different disk head to access a different platter surface. A track switch is the time it takes to move a disk head from the last track of a cylinder to the first one of the next.

The disk appears to its client as a linear array of logical blocks; these logical blocks are then mapped to physical sectors on the platters. This indirection has the advantage that the disk can reorganize blocks to avoid bad sectors and to improve performance, but it has the disadvantage that the client does not know where a particular logical block is located. If a client wants to derive this mapping, there are multiple sources of complexity. First, different tracks have different numbers of sectors; specifically, due to zoning, tracks near the outside of a platter have more sectors (and subsequently deliver higher bandwidth) than tracks near the spindle. Second, consecutive sectors across track and cylinder boundaries are skewed to adjust for head and track switch times; the skewing factor differs across zones as well. Third, flawed sectors are remapped through sparing; sparing may be done by remapping a bad sector (or track) to a fixed alternate location or by slipping the sector (or track) and all subsequent ones to the next sector (or track).

3.1.2 Input Parameters

As previously explained, read and write operations take different times to execute. In addition, the type of the last operation issued also influences service time [4, 18]. To account for these factors in our table-based model, we record the request type (read or write) of the current and previous requests as one of the input parameters.

The other input parameter is the inter-request distance between logical block addresses, which captures some

of the aforementioned underlying characteristics of the disk, while missing others. We note that ordering requests based on the time for a given distance is significantly different than using the distance itself. Due to the complexity of disk geometry, some requests that are separated by a larger logical distance can be positioned more rapidly; the relationship between the logical block address distance and positioning time is not linear.

In the opinion of Ruemmler and Wilkes [18], the following aspects of the disk should be modeled for the best accuracy: seek time (calculated with two separate functions depending upon the seek distance from the current and final cylinder position of the disk head and different for reads and writes), head and track switches, rotation latency, data layout (including reserved sparing areas, zoning, and track and cylinder skew), and data caching (both read-ahead and write-behind). We briefly discuss the extent to which each of these components is captured with our approach.

Our approach accounts for the combined costs of seek time, head and track switches, and rotation layout, but in a probabilistic manner. That is, for a given inter-request distance, there is some probability that a request crosses track or even cylinder boundaries. Requests of a given distance that cross the same number of boundaries have the same total positioning time: the same number of track seeks, the same number of head and/or track switches, and the same amount of rotation.

We note that the table-based method for tracking positioning time can be *more* accurate than that advocated by Ruemmler and Wilkes; instead of expressing positioning time as a value computed as a sum of functions (seek time, rotation time, caching, etc.), the Disk Mimic records the precise positioning time for each distance.

The cost incurred by the rotation of the disk has two components: the rotational distance between the previous and current request, and the elapsed time between the two requests (and thus, the amount of rotation that has already occurred). Although using inter-request distance probabilistically captures the rotational distance, the Disk Mimic does not record the amount of time that has elapsed since the last request. This omission is not an issue for disk scheduling in the presence of a full queue of requests; in this case, the inter-arrival time between requests at the disk is negligible and, thus, can be ignored. Ignoring time causes inaccuracies when scheduling the first request after an idle period; however, if the disk is often idle, then I/O scheduling is not an important problem.

Data layout is incorporated fairly well by the Disk Mimic as well. The number of sectors per track and number of cylinders impact our measured values in that these sizes determine the probability that a request of a given inter-request distance crosses a boundary; thus, these sizes impact the probability of each observed time in the distri-

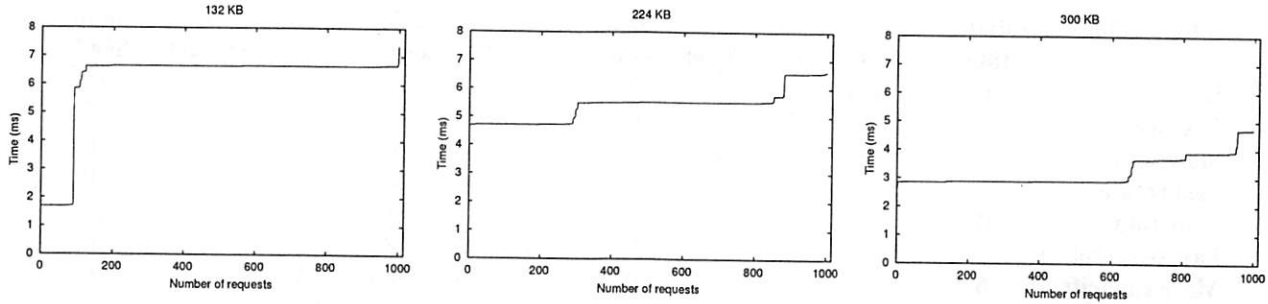


Figure 1: **Distribution of Off-Line Probe Times for Three Inter-Request Distances.** Each graph shows a different inter-request distance: 132 KB, 224 KB, and 300 KB. Along the *x*-axis, we show each of the 1000 probes performed (sorted by time) and along the *y*-axis we show the time taken by that probe. These times are for the IBM 9LZX disk.

bution. Although zoning behavior and bad sectors are not tracked by our model, previous research has shown that this level of detail does not help with scheduling [31].

The aspect which we model the least directly is that of general caching. However, the Disk Mimic will capture the effects of simple prefetching, which is the most important aspect of caching for scheduling [31]. For example, if a read of one sector causes the entire track to be cached, then the Disk Mimic will observe the faster performance of accesses with distances less than that of a track. In this respect, configuring the Disk Mimic on-line by observing the actual workload could be more accurate than configuring off-line, since the locality of the workload is captured.

Given the complexity associated with the inter-request distance, we concentrate on the issues related to this input parameter. For different values of the request type, the output of the Disk Mimic has the same characteristics, and thus we do not need to explore all the possible combinations of the two input parameters in our further discussions. Hence when we refer to inter-request distance we assume the request type is fixed.

3.2 Results

To illustrate some of the complexity of using inter-request distance as predictor of request time, we show the distribution of times observed. For these experiments, we configure the Disk Mimic off-line as follows.

The Disk Mimic configures itself by probing the I/O device using fixed-size requests (e.g., 1 KB). For each of the possible inter-request distances covering the disk (both negative and positive), the Disk Mimic samples a number of points of the same distance: it accesses a block the specified distance from the previous block. To avoid any caching or prefetching performed by the disk, the Disk Mimic accesses a random location before each new probe of the required distance. The observed times are recorded in a table, indexed by the inter-request distance and the corresponding operation type.

In Figure 1 we show a small subset of the data collected on an IBM 9LZX disk. The figure shows the distribution

of 1000 samples for three inter-request distances of 132 KB, 224 KB, and 300 KB. In each case, the *y*-axis shows the request time of a sample and the points along the *x*-axis represent each sample, sorted by increasing request time.

We make two important observations from the sampled times. First, for a given inter-request distance, the observed request time is not constant; for example, at a distance of 132 KB, about 10% of requests require 1.8 *ms*, about 90% require 6.8 *ms*, and a few require almost 8 *ms*. Given this multi-modal behavior, the time for a single request cannot be reliably predicted from only the inter-request distance; thus, one cannot usually predict whether a request of one distance will be faster or slower than a request of a different distance. Nevertheless, it is often possible to make reasonable predictions based upon the probabilities: for example, from this data, one can conclude that a request of distance 132 KB is likely to take longer than one of 224 KB.

Second, from examining distributions for different inter-request distances, one can observe that the number of transitions and the percentage of samples with each time value varies across inter-request distances. The number of transitions in each graph corresponds roughly to the number of track (or cylinder) boundaries that can be crossed for this inter-request distance.

This data shows that a number of important issues remain regarding the configuration of the Disk Mimic. First, since there may be significant variation in request times for a single inter-request distance, what summary metric should be used to summarize the distribution? Second, how many samples are required to adequately capture the behavior of this distribution? Third, must each inter-request distance be sampled, or is it possible to interpolate intermediate distances? We investigate these issues in Section 5.

4 Methodology

To evaluate the performance of SMTF scheduling, we consider a range of disk drive technology, presented in

Configuration	rotation time	1 cyl	seek 400	3000	head switch	cyl switch	track skew	cyl skew	sectors per track	num heads
1 Base	6	0.8	6.0	8	0.79	1.78	36	84	272	10
2 Fast seek	6	0.16	1.32	1.6	0.79	1.00	36	46	272	10
3 Slow seek	6	2.0	33.0	40.0	0.79	2.80	36	127	272	10
4 Fast rotate	2	0.8	6.0	8	0.79	1.78	108	243	272	10
5 Slow rotate	12	0.8	6.0	8	0.79	1.78	18	41	272	10
6 Fast seek+rot	2	0.160	1.32	1.6	0.79	1.00	108	136	272	10
7 More capacity	6	0.8	6.0	8	0.79	1.78	36	84	544	20
8 Less capacity	6	0.8	6.0	8	0.79	1.78	36	84	136	5

Table 1: **Disk Characteristics.** Configurations of eight simulated disks. Times for rotation, seek, and head and cylinder switch are in milliseconds, the cylinder and track skews are expressed in sectors. In most experiments, the base disk is used.

Table 1. We have implemented a disk simulator that accurately models seek time, fixed rotation latency, track and cylinder skewing, and a simple segmented cache. The first disk, also named the *base disk*, simulates a disk with performance characteristics similar to an IBM 9LZX disk. The seek times, cache size and number of segments, head and cylinder switch times, track and cylinder skewing and rotation times are either measured by issuing SCSI commands and measuring the elapsed time, or directly querying the disk, similar to the approach used by Schindler and Ganger [19], or by using the values provided by the manufacturer. The curve corresponding to the seek time is modeled by probing an IBM 9LZX disk for a range of seek distances (measured as the distance in cylinders from the previous cylinder position to the current one) and then curve fitting the values to use the two-function equation proposed by Ruemmler and Wilkes [18]. For short seek distances the seek time is proportional to the square root of the cylinder distance, and for longer distances the seek time is proportional to the cylinder distance. The middle value in the seek column represents the cylinder distance where the switch between the two functions occurs. For example, for the base disk, if the seek distance is smaller than 400 cylinders, we use the square root function.

For the other disk configurations we simulate, we start from the base disk and vary different parameters that influence the positioning time. For example, disk configuration number 2 (*Fast seek*) represents a disk that has a fast seek time and the numbers used to compute the seek curve are adjusted accordingly, as well as the number of sectors that constitute the cylinder skew. Similarly for disk configuration number 4 (*Fast rotate*) the time to execute a rotation is decreased by a factor of three and the number of track and cylinder skew sectors are increased. The other disk configurations account for disks that have a slower seek time, slower rotation time, faster seek time, faster rotation time and more or less capacity than the base disk. In addition to using the described simulated disks we also run our experiments on an IBM 9LZX disk.

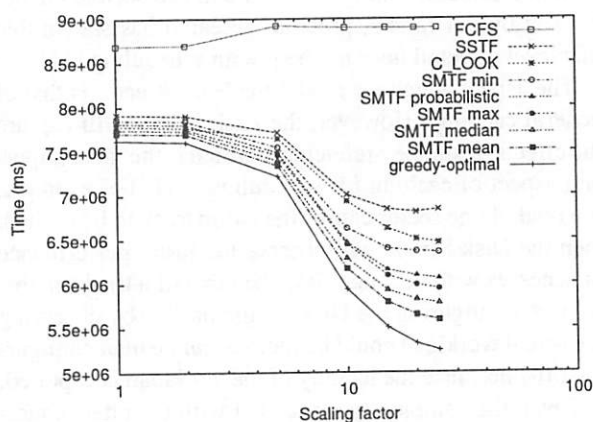


Figure 2: **Sensitivity to Summary Metrics.** This graph compares the performance of a variety of scheduling algorithms on the base simulated disk and the week-long HP trace. For the SMTF schedulers, no interpolation is performed and 100 samples are obtained for each data point. The x-axis shows the compression factor applied to the workload. The y-axis reports the time spent at the disk.

To evaluate scheduling performance, we show results from a set of traces collected at HP Labs [17]; in most cases, we focus on the trace for the busiest disk from the week of 5/30/92 to 6/5/92. For our performance metric, we report the time the workload spent at the disk. To consider the impact of heavier workloads and longer queue lengths, we compress the inter-arrival time between requests. When scaling time, we attempt to preserve the dependencies across requests in the workload by observing the blocks being requested; we assume that if a request is repeated to a block that has not yet been serviced, that this request is dependent on the previous request first completing. Thus, we hold repeated requests, and all subsequent requests, until the previous identical request completes.

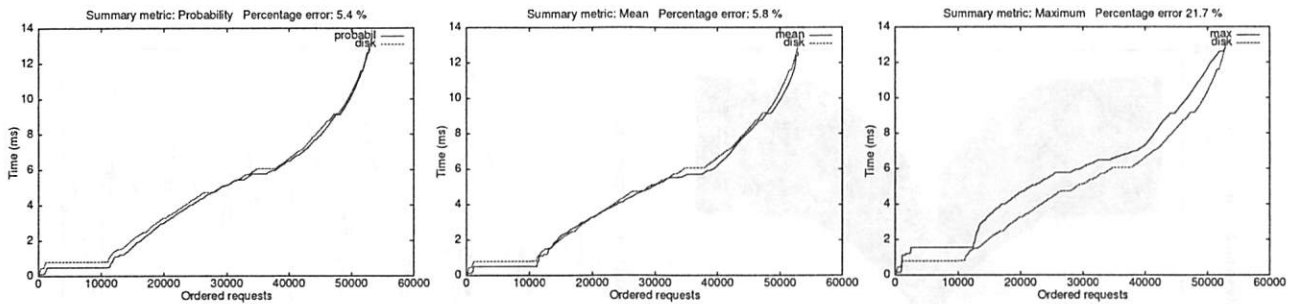


Figure 3: **Demerit Figures for SMTF with Probability, Mean, and Maximum Summary Metrics.** Each graph shows the demerit figure for a different summary metric. These distributions correspond to the one day from the experiments shown in Figure 2 with a compression factor of 20.

5 Off-Line Configuration

The SMTF scheduler can be configured both on-line and off-line. We now explore the case when the Disk Mimic has been configured off-line; again, although the Disk Mimic is configured off-line, the simulation and predictions required by the scheduler are still performed on-line within the system. As described previously, configuring the Disk Mimic off-line involves probing the underlying disk with requests that have a range of inter-request distances. We note that even when the model is configured off-line, the process of configuring SMTF remains entirely automatic and portable across a range of disk drives. The main drawback to configuring the Disk Mimic off-line is a longer installation time when a new device is added to the system: the disk must be probed before it can be used for workload traffic.

5.1 Summary Data

To enable the SMTF scheduler to easily compare the expected time of all of the requests in the queue, the Disk Mimic must supply a summary value for each distribution as a function of the inter-request distance. Given the multi-modal characteristics of these distributions, the choice of a summary metric is not obvious. Therefore, we evaluate five different summary metrics: mean, median, maximum, minimum, and probabilistic, which randomly picks a value from the sampled distribution according to its probability.

The results for each of these summary metrics on the base simulated disk are shown in Figure 2. For the workload, we consider the week-long HP trace, scaled by the compression factor noted on the x -axis. The graph shows that FCFS, SSTF, and C-LOOK all perform worse than each of the SMTF schedulers; as expected, the SMTF schedulers perform worse than the greedy-optimal scheduler, but the best approach is always within 7% for this workload. These results show that using inter-request distance to predict positioning time merits further attention.

Comparing performance across the different SMTF approaches, we see that each summary metric performs

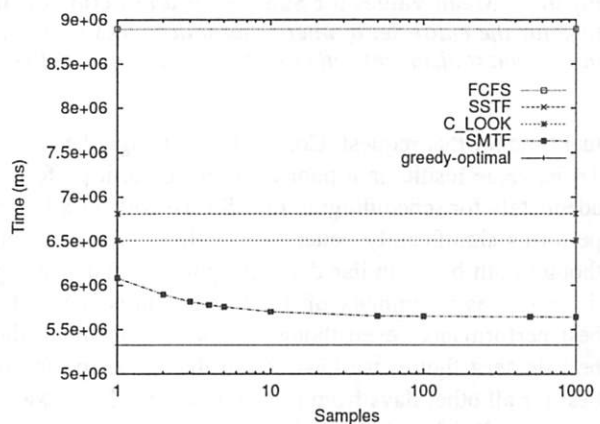


Figure 4: **Sensitivity to Number of Samples.** The graph shows that the performance of SMTF improves with more samples. The results are on the simulated disk and the week-long HP trace with a compression factor of 20. The x -axis indicates the number of samples used for SMTF. The y -axis shows the time spent at the disk.

quite differently. The ordering of performance from best to worse is: mean, median, maximum, probabilistic, and minimum. It is interesting to note that the scheduling performance of each summary metric is not correlated with its accuracy. The accuracy of disk models is often evaluated according to its *demerit figure* [18], which is defined as the root mean square of the horizontal distance between the time distributions for the model and the real disk. This point is briefly illustrated in Figure 3, which shows the distribution of actual times versus the predicted times for three different metrics: probabilistic, mean, and maximum.

As expected, the probabilistic model has the best demerit figure; with many requests, the distribution it predicts is expected to match that of the real device. However, the probabilistic model performs relatively poorly within SMTF because the time it predicts for any one request may differ significantly from the ac-

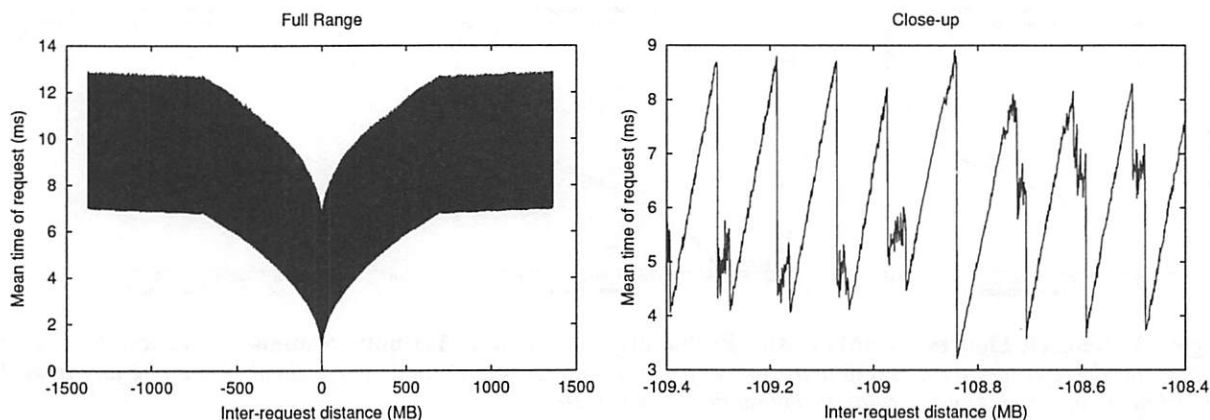


Figure 5: **Mean Values for Samples as a Function of Inter-request Distance.** The graph on the left shows the mean time for the entire set of inter-request distances on our simulated disk. The graph on the right shows a close-up for inter-request distances; other distances have qualitatively similar saw-tooth behavior.

tual time for that request. Conversely, although the maximum value results in a poor demerit figure, it performs adequately for scheduling; in fact, SMTF with maximum performs significantly better than with minimum, even though both have similar demerit figures. Finally, using the mean as a summary of the distribution achieves the best performance, even though it does not result in the best demerit figure; we have found that mean performs best for all other days from the HP traces we have examined as well. Thus, for the remainder of our experiments, we use the mean of the observed samples as the summary data for each inter-request distance.

5.2 Number of Samples

Given the large variation in times for a single inter-request distance, the Disk Mimic must perform a large number of probe samples to find the true mean of the distribution. However, to reduce the time required to configure the Disk Mimic off-line, we would like to perform as few samples as possible. Thus, we now evaluate the impact of the number of samples on SMTF performance.

Figure 4 compares the performance of SMTF as a function of the number of samples to the performance of FCFS, C-LOOK, SSTF, and optimal. As expected, the performance of SMTF increases with more samples; on this workload and disk, the performance of SMTF continues to improve up to approximately 10 samples. However, most interestingly, even with a single sample for each inter-request distance, the Disk Mimic performs better than FCFS, C-LOOK, and SSTF.

5.3 Interpolation

Although the number of samples performed for each inter-request distance impacts the running time of the off-line probe process, an even greater issue is whether each distance must be explicitly probed or if some can be inter-

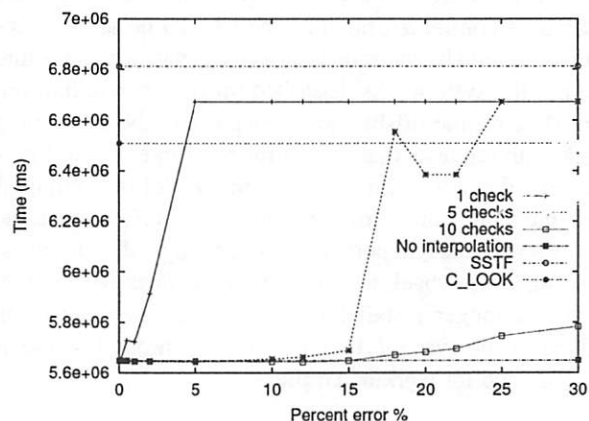


Figure 6: **Sensitivity to Interpolation.** The graph shows performance with interpolation as a function of the percent of allowable error. Different lines correspond to different numbers of check points, N . The x-axis is the percent of allowable error and the y-axis is the time spent at the disk. These results use the base simulated disk and the week-long HP trace with a compression factor of 20.

polated from other distances. Due to the large number of potential inter-request distances on a modern storage device (*i.e.*, two times the number of sectors for both negative and positive distances), not only does performing all of the probes take a significant amount of time, but storing each of the mean values is prohibitive as well. For example, given a disk of size 10 GB, the amount of memory required for the table can exceed 800 MB. Therefore, we explore how some distances can be interpolated without making detailed assumptions about the underlying disk.

To illustrate the potential for performing simple interpolations, we show the mean value as a function of the inter-request distance in Figure 5. The graph on the left

Check Points N	Acceptable Error
1	1 %
2	2 %
3	5 %
4	10 %
5	15 %
10	20 %

Table 2: Allowable Error for Interpolation. *The table summarizes the percentage within which an interpolated value must be relative to the probed value in order to infer that the interpolation is successful. As more check points are performed between two inter-request distances, the allowable error increases. The numbers were gathered by running a number of different workloads on the simulated disks and observing the point at which performance with interpolation degrades relative to that with no interpolation.*

shows the mean values for all inter-request distances on our simulated disk. The curve of the two bands emanating from the middle point corresponds to the seek curve of the disk (*i.e.*, for short seeks, the time is proportional to the square root of the distance, whereas for long, the time is linear with distance); the width of the bands is relatively constant and corresponds to the rotation latency of the disk. The graph on the right shows a close-up of the inter-request distances. This graph shows that the times follow a distinct saw-tooth pattern; as a result, a simple linear model can likely be used to interpolate some distances, but care must be taken to ensure that this model is applied to only relatively short distances.

Given that the length of the linear regions varies across different disks (as a function of the track and cylinder size), our goal is not to determine the particular distances that can be interpolated successfully. Instead, our challenge is to determine when an interpolated value is “close enough” to the actual mean such that scheduling performance is impacted only negligibly.

Our basic off-line interpolation algorithm is as follows. After the Disk Mimic performs S samples of two inter-request distances *left* and *right*, it chooses a random distance *middle* between *left* and *right*; it then linearly interpolates the mean value for *middle* from the means for *left* and *right*. If the interpolated value for *middle* is within *error* percent of the probed value for *middle*, then the interpolation is considered successful and all the distances between *left* and *right* are interpolated. If the interpolation is not successful, the Disk Mimic recursively checks the two smaller ranges (*i.e.*, the distances between *left* and *middle* and between *middle* and *right*) until either the intermediate points are successfully interpolated or until all points are probed.

For additional confidence that linear interpolation is

valid in a region, we consider a slight variation in which N points between *left* and *right* are interpolated and checked. Only if all N points are predicted with the desired level of accuracy is the interpolation considered successful. The intuition of performing more check points is that a higher error rate can be used and interpolation can still be successful.

Figure 6 shows the performance of SMTF when distances are interpolated; the graph shows the effect of increasing the number of intermediate points N that are checked, as well as increasing the acceptable error, *error*, of the interpolation. We make two observations from this graph.

First, SMTF performance decreases as the allowable error of the check points increases. Although this result is to be expected, we note that performance decreases dramatically with the error not because the error of the checked distances is increased, but because the interpolated distances are inaccurate by much more. For example, with a single check point (*i.e.*, $N = 1$) and an error level of 5%, we have found that only 20% of the interpolated values are actually accurate to that level and the average error of all interpolated values increases to 25% (not shown). In summary, when error increases significantly, there is not a linear relationship for the distances between *left* and *right* and interpolation should not be performed.

Second, SMTF performance for a fixed error increases with the number of intermediate check points N . The effect of performing more checks is to confirm that linear interpolation across these distances is valid. For example, with $N = 10$ check points and *error* = 5%, almost all interpolated points are accurate to that level and the average error is less than 1% (also not shown).

Table 2 summarizes our findings for a wider number of check points. The table shows the allowable error percentage as a function of the number of check points, N , to achieve scheduling performance that is very similar to that with all probes. Thus, the final probe process can operate as follows. If the interpolation of one distance between *left* and *right* has an error less than 1%, it is deemed successful. Otherwise, if two distances between *left* and *right* have errors less than 2%, the interpolation is successful as well. Thus, progressively more check points can be made with higher error rates to be successful. With this approach, 90% of the distances on the disk are interpolated instead of probed, and yet scheduling performance is virtually unchanged; thus, interpolation leads to a 10-fold memory savings.

5.4 Disk Characteristics

To demonstrate the robustness and portability of the Disk Mimic and SMTF scheduling, we now consider the full range of simulated disks described in Table 1. The performance of FCFS, C-LOOK, SSTF, and SMTF relative

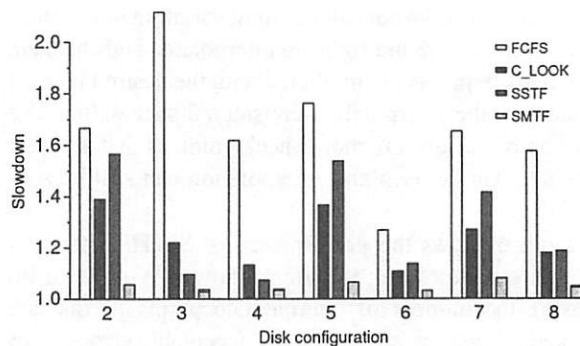


Figure 7: **Sensitivity to Disk Characteristics.** This figure explores the sensitivity of scheduling performance to the disk characteristics shown in Table 1. Performance is shown relative to greedy-optimal. We report values for SMTF using interpolation. The performance of SMTF without interpolation (i.e., all probes) is very similar.

to greedy-optimal for each of the seven new disks is summarized in Figure 7. We show the performance of SMTF with interpolation. The performance of SMTF with and without interpolation is nearly identical. As expected, FCFS performs the worst across the entire range of disks, sometimes performing more than a factor of two slower than greedy-optimal. C-LOOK and SSTF perform relatively well when seek time dominates performance (e.g., disks 3 and 4); SSTF performs better than C-LOOK in these cases as well. Finally, SMTF performs very well when rotational latency is a significant component of request positioning (e.g., disks 2 and 5). In summary, across this range of disks, SMTF always performs better than both C-LOOK and SSTF scheduling and within 8% of the greedy-optimal algorithm.

To show that SMTF can handle the performance variation of real disks, we compare the performance of our implementation of SMTF to that of C-LOOK when run on the IBM 9LZX disk. On the one week HP trace, we achieve a performance improvement of 8% for SMTF compared C-LOOK and an improvement of 12% if idle time is removed from the trace. This performance improvement is not as significant as it could be for two reasons. First, the IBM 9LZX disk has a relatively high ratio of seek to rotation time; the performance improvement of SMTF relative to C-LOOK is greater when rotation time is a more significant component of positioning. Second, the HP trace exercises a large amount of data on the disk; when the locality of the workload is low as in this trace, seek time further dominates positioning time.

To explore the effect of workload locality we create a synthetic workload of random 1 KB reads and writes with no idle time; the maximum inter-request distance is varied, as specified on the x -axis of Figure 8. This graph shows that the performance improvement of SMTF rela-

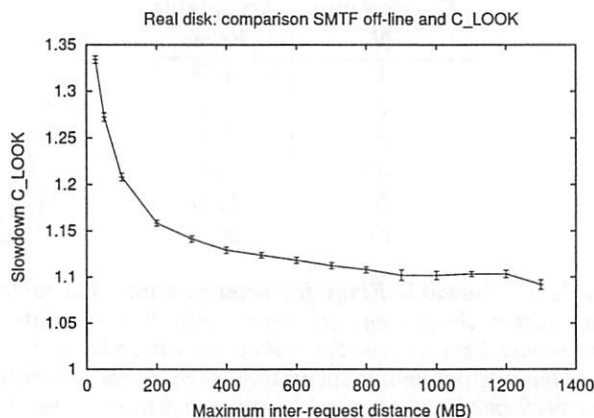


Figure 8: **Real Disk Performance.** This graph shows the slowdown of C-LOOK when compared to the SMTF configured off-line. The workload is a synthetically generated trace and the numbers are averages over 20 runs. The standard deviation is also reported. The x -axis shows the maximum inter-request distance existent in the trace and the y -axis reports the percentage slowdown of the C-LOOK algorithm.

tive to C-LOOK varies between 32% and 8% as the inter-request distance varies from 25 MB to 1.3 GB. Given that most file systems (e.g., Linux ext2) try to optimize locality by placing related files in the same cylinder group, SMTF can optimize accesses better than C-LOOK in practice. Thus, we believe that SMTF is a viable option for scheduling on real disks.

6 On-Line Configuration

We now explore the SMTF scheduler when all configuration is performed on-line. With this approach, there is no overhead at installation time to probe the disk drive; instead, the Disk Mimic observes the behavior of the disk as the workload runs. As in the off-line version, the Disk Mimic records the observed disk times as a function of its inter-request distance, but in this case it has no control over the inter-request distances it observes.

6.1 General Approach

For the on-line version, we assume that many of the lessons learned from off-line configuration hold. First, we continue to use the mean to represent the distribution of times for a given inter-request distance. Second, we continue to rely upon interpolation; note that when the Disk Mimic is configured on-line, interpolation is useful not only for saving space, but also for providing new information about distances that have not been observed.

The primary challenge that SMTF must address in this situation is how to schedule requests when some of the inter-request distances have unknown times (i.e., this inter-request distance has not yet been observed by the

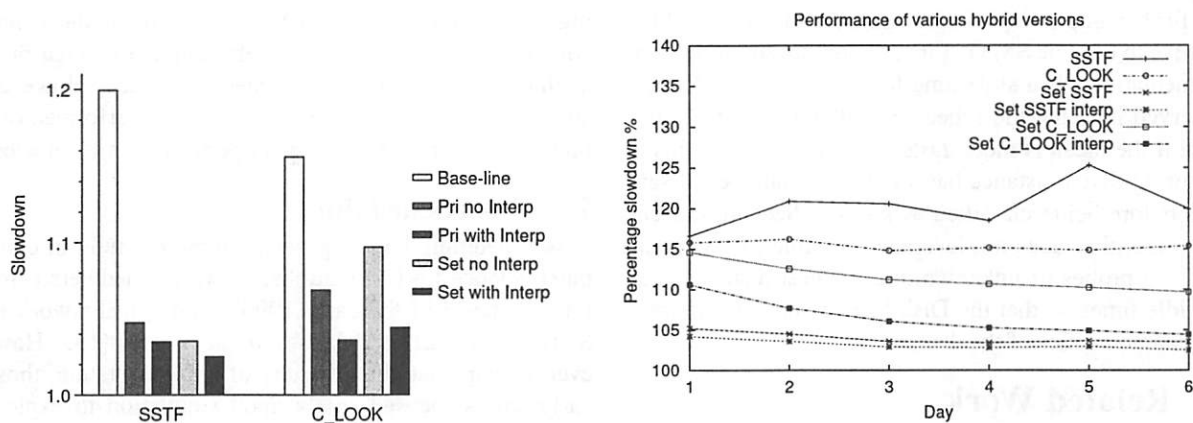


Figure 9: **Performance of On-Line SMTF.** The first graph compares the performance of different variations of on-line SMTF; the performance of the last day of the week-long HP trace is shown relative to off-line SMTF. The second graph shows that the performance of Online-Set improves over time as more inter-request distances are observed.

Disk Mimic and the Disk Mimic is unable to confirm that it can be interpolated successfully). We consider two algorithms for comparison. Both algorithms assume that there is a base scheduler (either C-LOOK or SSTF) which is used when the Disk Mimic does not have sufficient information.

The first algorithm, *Online-Priority*, schedules only those requests for which the Disk Mimic has information. Specifically, *Online-Priority* gives strict priority to those requests in the queue that have an inter-request distance with a known time; among those requests with known times, the request with the minimum mean time is picked. With *Online-Priority*, the base scheduler (e.g., C-LOOK or SSTF) is only used when no inter-request distances for the current queue are known. There are two problems with this approach. First, given its preference for scheduling already known inter-request distances, *Online-Priority* may perform worse than its base scheduler. Second, schedules with a diversity of distances may never be produced and thus the Disk Mimic may never observe some of the most efficient distances.

The second algorithm, *Online-Set*, improves on both of these limitations by using the decision of the base scheduler as its starting point, and scheduling a different request only when the Disk Mimic has knowledge that performance can be improved. Specifically, *Online-Set* first considers the request that the base scheduler would pick. If the time for the corresponding distance is not known by the Disk Mimic, then this request is scheduled. However, if the time is known, then all of the requests with known inter-request distances are considered and the one with the fastest mean is chosen. Thus, *Online-Set* should only improve on the performance of the base scheduler and it is likely to schedule a variety of inter-request distances when it is still learning.

6.2 Experimental Results

To evaluate the performance of the on-line algorithms, we return to the base simulated disk. The left-most graph of Figure 9 compares the performance of *Online-Priority* and *Online-Set*, when either C-LOOK or SSTF is used as the baseline algorithm and both with and without interpolation. Performance is expressed in terms of slowdown relative to the off-line version of SMTF. We make three observations from this graph.

First, and somewhat surprising, although C-LOOK performs better than SSTF for this workload and disk, SMTF performs noticeably better with SSTF than with C-LOOK as a base; with C-LOOK, the Disk Mimic is not able to observe inter-request distances that are negative (i.e., backward) and thus does not discover distances that are close together. Second, *Online-Set* performs better than *Online-Priority* with SSTF as the base scheduler. Third, although interpolation does significantly improve the performance of *Online-Priority* and of *Online-Set* with C-LOOK, it leads to only a small improvement with *Online-Set* and SSTF. Thus, as with off-line configuration, the primary benefit of interpolation is to reduce the memory requirements of the Disk Mimic, as opposed to improving performance.

The right-most graph of Figure 9 illustrates how the performance of *Online-Set* improves over time as more inter-request distances are observed. We see that the performance of the *Online-Set* algorithms (with and without interpolation) is better than the base-line schedulers of SSTF and C-LOOK even after one day of the original trace (i.e., approximately 150,000 requests). The performance of *Online-Set* with SSTF converges to within 3% of the off-line version after four days, or only about 750,000 requests.

At this point, we feel that there are two opportunities

for further improving the performance of on-line SMTF relative to off-line SMTF. First, in our current on-line implementations, if a slow time for a particular distance is observed initially, the scheduler will avoid that distance even if the mean is much faster. One can address this by requiring that a distance has a minimum number of samples before being classified as known. Second, our current algorithm does not leverage idle time [6]. One can perform probes of unknown inter-request distances during idle times so that the Disk Mimic can learn more of the characteristics of the disk.

7 Related Work

The approach we propose brings together two areas of study: disk modeling and disk scheduling. We present related work in both areas and compare it to our method.

7.1 Disk Modeling

The classic paper describing models of disk drives is that by Ruemmler and Wilkes [18]. The main focus of this work is to enable an informed trade-off between simulation effort and the resulting accuracy of the model. Ruemmler and Wilkes evaluate the aspects of a disk that should be modeled for a high level of accuracy, using the *demerit figure*. Other researchers have noted that additional non-trivial assumptions must be made to model disks to the desired accuracy level [14]; modeling cache behavior is a particularly challenging aspect [24].

Given that the detailed knowledge for modeling disks is not available from documentation, researchers have developed innovative methods to acquire the information. For example, Worthington *et al.* describe techniques for SCSI drives that extract time parameters such as the seek curve, rotation speed, and command overheads as well as information about the data layout on disk and the caching and prefetching characteristics [32]; many of these techniques are automated in later work [19].

Modeling storage devices using tables of past performance has also been explored in previous work; in most previous work [1, 7], high-level system parameters (*e.g.*, load, number of disks, and operation type) are used as indices into the table. Anderson [1] also uses the results on-line, to assist in the reconfiguration of disk arrays. The approach most similar to ours is that of Thornock *et al.* [27]. In this work, the authors use stochastic methods to build a model of the underlying drive. However, the application of this model is to standard, off-line simulation; specifically, the authors study block reorganization, similar to earlier work by Ruemmler and Wilkes [16].

At a higher level, Seltzer and Small suggest *in situ* simulation as a method for building more adaptive operating systems [22]. In this work, the authors suggest that operating systems can utilize in-kernel monitoring and adaptation to make more informed policy decisions. By trac-

ing application activity, the VINO system can determine whether the current policy is behaving as expected or if another policy should be switched into place. However, actual simulations of system behavior are performed off-line, as a “last resort” when poor performance is detected.

7.2 Disk Scheduling

Disk scheduling has long been a topic of study in computer science [29]. Rotationally-aware schedulers came into existence in the early 1990’s, through the work of Seltzer *et al.* [21] and Jacobson and Wilkes [13]. However, perhaps due the difficulty of implementation, those early works focused solely upon simulation to explore the basic ideas. Only recently have implementations of rotationally-aware schedulers been described within the literature, and those are crafted with extreme care [11, 33].

More recently, Worthington *et al.* [31] examine the benefits of even more detailed knowledge of disk drives within OS-level disk schedulers. They find that algorithms that mesh well with the modern prefetching caches perform best, but that detailed logical-to-physical mapping information is not currently useful.

Anticipatory scheduling is a recent scheduling development that is complementary to our on-line simulation-based approach [12]. An anticipatory scheduler makes the assumption that there is likely to be locality in a stream of requests from a given process; by waiting for the next request (instead of servicing a request from a different process), performance can be improved. The authors also note the difficulty of building a rotationally-aware scheduler, and instead use an empirically-generated curve-fitted estimate of disk access-time costs; the Disk Mimic would yield a performance benefit over this simplified approach.

8 Conclusions

In this paper, we have explored some of the issues of using simulation within the system to make run-time scheduling decisions; in particular, we have focused on how a disk simulator can automatically model a range of disks without human intervention. We have shown that the Disk Mimic can model the time of a request by simply observing the request type and the logical distance from the previous request and predicting that it will behave similarly to past requests with the same parameters. The Disk Mimic can configure itself for a given disk by either probing the disk off-line or, at a slight performance cost, by observing requests sent to the disk on-line. We have demonstrated that a shortest-mimicked-time-first (SMTF) disk scheduler can significantly improve disk performance relative to FCFS, SSTF, and C-LOOK for a range of disk characteristics.

In the future, we plan to show that SMTF scheduling is appropriate for a range of storage devices other than disk drives. For example, RAID systems [15], network-

attached storage devices [5], MEMS-based devices [20], tapes [9], and non-volatile memory [28] may all be used as building blocks in a storage system. Each of these devices has its own complex performance characteristics and it would be ideal if the I/O scheduler could automatically adapt to any of these devices.

Acknowledgments

We would like to thank Nathan Burnett, Timothy Denehy, Brian Forney, and Muthian Sivathanu for their feedback on this paper. We also would like to thank Vern Paxson, our shepherd, and the anonymous reviewers for their thoughtful suggestions, all of which have greatly improved the content of this paper. Finally, we thank the Computer Systems Lab for their tireless assistance in providing a terrific environment for computer science research. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, an IBM Faculty Award, and the Wisconsin Alumni Research Foundation.

References

- [1] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-04, HP Laboratories, July 2001.
- [2] M. Andrews, M. Bender, and L. Zhang. New algorithms for the disk scheduling problem. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 550-559, 1996.
- [3] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *The 18th Symposium on Operating Systems Principles (SOSP)*, pages 43-56, October 2001.
- [4] G. R. Ganger, B. L. Worthington, and Y. N. Patt. The DiskSim Simulation Environment - Version 2.0 Reference Manual. <http://citeseer.nj.nec.com/article/ganger99disksim.html>.
- [5] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server Scaling with Network-Attached Secure Disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272-284, Seattle, WA, June 1997.
- [6] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is Not Sloth. In *Proceedings of the Winter '95 USENIX Technical Conference*, pages 201-212, New Orleans, Louisiana, January 1995.
- [7] C. Gotlieb and G. MacEwen. Performance of movable-head disk storage devices. *Journal of the Association for Computing Machinery*, 20(4):604-623, 1973.
- [8] J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, and G. R. Ganger. Timing-accurate Storage Emulation. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 75-88, Monterey, CA, January 2002.
- [9] B. Hillyer and A. Silberschatz. On the modeling and performance characteristics of a serpentine tape drive. In *Proceedings of 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 170-179, 1996.
- [10] M. Hofri. Disk scheduling: FCFS vs. SSTF revisited. *Communications of the ACM*, 23(11):645-653, 1980.
- [11] L. Huang and T. Chiueh. Implementation of a rotation latency sensitive disk scheduler. Technical Report ECSL-TR81, SUNY, Stony Brook, March 2000.
- [12] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, pages 117-130, October 2001.
- [13] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7, HP Laboratories, 1991.
- [14] D. Kotz, S. B. Toh, and S. Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical Report TR94-220, Dartmouth College, 1994.
- [15] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):109-116, September 1988.
- [16] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-91-156, Hewlett Packard Laboratories, October 1991.
- [17] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 405-420, 1993.
- [18] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17-28, 1994.

- [19] J. Schindler and G. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, November 1999.
- [20] S. W. Schlosser, J. L. Griffin, D. Nagle, and G. R. Ganger. Designing computer systems with MEMS-based storage. In *Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2000.
- [21] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, CA, 1990.
- [22] M. I. Seltzer and C. Small. Self-Monitoring and Self-Adapting Systems. In *Proceedings of the 1997 Workshop on Hot Topics on Operating Systems*, pages 124–129, Chatham, MA, May 1997.
- [23] P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *Proceedings of the 1998 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 44–55, June 1998.
- [24] E. A. M. Shriver, A. Merchant, and J. Wilkes. An analytic behavior model for disk drives with read-ahead caches and request reordering. In *Proceedings of 1998 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 182–191, 1998.
- [25] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.
- [26] T. J. Teorey and T. B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3):177–184, 1972.
- [27] N. C. Thornock, X.-H. Tu, and J. K. Flanagan. A stochastic Disk I/O Simulation Technique. In *Proceedings of the 1997 Winter Simulation Conference*, pages 1079–1086, 1997.
- [28] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: better performance through a disk/persistent-RAM hybrid file system. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 15–28, Monterey, CA, June 2002.
- [29] N. C. Wilhelm. An anomaly in disk scheduling: a comparison of FCFS and SSTF seek scheduling using an empirical model for disk accesses. *Communications of the ACM*, 19(1):13–17, 1976.
- [30] J. Wilkes. The Pantheon storage-system simulator. Technical Report HPL-SSP-95-14, HP Laboratories, Palo Alto, CA, December 1995.
- [31] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 241–251, Nashville, TN, USA, 1994.
- [32] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. Technical Report CSE-TR-323-96, Carnegie Mellon University, 1996.
- [33] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation*, pages 243–258, San Diego, 2000. USENIX Association.

Controlling your PLACE in the File System with Gray-box Techniques

James A. Nugent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

*Computer Sciences Department
University of Wisconsin, Madison*

Abstract

We present the design and implementation of PLACE, a gray-box library for controlling file layout on top of FFS-like file systems. PLACE exploits its knowledge of FFS layout policies to let users place files and directories into specific and localized portions of disk. Applications can use PLACE to colocate files that exhibit temporal locality of access, thus improving performance. Through a series of microbenchmarks, we analyze the overheads of controlling file layout on top of the file system, showing that the overheads are not prohibitive, and also discuss the limitations of our approach. Finally, we demonstrate the utility of PLACE through two case studies: we demonstrate the potential of file layout rearrangement in a web-server environment, and we build a benchmarking tool that exploits control over file placement to quickly extract low-level details from the disk system. In the traditional gray-box manner, the PLACE library achieves these ends entirely at user level, without changing a single line of operating system source code.

1 Introduction

Creators of high-performance I/O-intensive applications, including database management systems and web servers, have long yearned for control over the placement of their data on disk [26]. Proper data allocation can exploit locality of access within a particular workload, increasing disk efficiency and thereby improving overall performance.

However, many file systems do not provide the explicit controls that are needed by applications to affect their desired file layouts. For example, UNIX file systems based on the Berkeley Fast File System (FFS) [13] group files by a set of heuristics, specifically trying to group inodes and data blocks of files that reside in the same directory. Applications that wish to have full control over layout traditionally have avoided using file systems altogether, thus relinquishing convenience for control.

Gray-box techniques [1, 4] are a promising approach

that can be used to gather information about and exert control over systems that do not export the necessary interfaces to do so. By treating a system as a *gray box*, one assumes some general knowledge of how the system behaves or is implemented; such knowledge, combined with run-time observations of the system, enables the construction of more powerful services than those exported by the base system.

In this paper, we explore the application of gray-box techniques to the file placement problem. Specifically, to retain the convenience of the file system while regaining control over placement, we introduce PLACE (Positional LAYout Controller), a system that exploits gray-box techniques to give applications improved control over file placement. The system is depicted in Figure 1.

The most important component of PLACE is the PLACE Information and Control Layer (ICL). The PLACE ICL allows applications to group files or directories into localized portions of the disk, specifically into a particular group. Proper placement of data can improve both read and write performance; by collocating files that are likely to be accessed at nearly the same time, applications can improve their performance by “short-stroking” the disk, *i.e.*, reducing the cost of seeks by limiting arm movement to a certain portion of the disk. Applications that do not use the PLACE library operate as expected.

The key to the PLACE implementation is the *shadow directory tree* (SDT). The SDT is a hidden control structure that the PLACE ICL uses to control where files are placed on disk. By carefully creating this structure and exploiting our gray-box knowledge of file system behavior, the SDT enables the PLACE ICL to place files according to user preferences in a correct and efficient manner. Creating and maintaining this structure is one of the central challenges in implementing PLACE.

We first evaluate the PLACE ICL with a set of microbenchmarks to understand the basic costs and potential benefits of using PLACE. We find that the costs of using PLACE are reasonable, although a controlled file or directory creation is more costly than standard versions

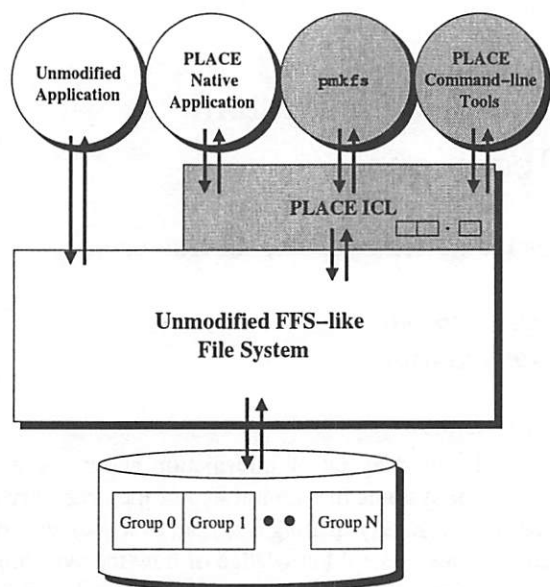


Figure 1: **The PLACE System.** The PLACE system consists of three components, highlighted in gray in the figure. The most important component is the PLACE Information and Control Layer (ICL), which uses gray-box techniques to discover information about the file system, and then exploits that knowledge to enable applications that link with it to control file and directory layout. The two other components of PLACE are the tool `pmkfs`, which is used to initialize the PLACE on-disk structures, and a set of PLACE command-line tools. PLACE currently works on top of “FFS-like” file systems by learning of their internal group structure and exposing this structure through the PLACE ICL.

of either operation in our prototype implementation. We also find that the potential benefits are substantial; random I/O performance improves dramatically when related data items are grouped into a small portion of the disk, and large files can be placed onto the outer tracks of a disk to improve throughput due to zoning effects [15].

We then demonstrate the utility of PLACE with two separate application studies. In the first, we show how a web server can use PLACE to group files that exhibit temporal access locality. Even when utilizing simple placement heuristics, collocation with PLACE improves web server throughput noticeably. In the second, we show how a high-speed user-level benchmarking tool called FAST can use PLACE to rapidly construct its testing infrastructure. Through controlled placement of files, FAST can extract important disk characteristics such as seek time and bandwidth in seconds.

The rest of this paper is organized as follows. In Section 2, we describe the design and implementation of PLACE, and in Section 3, we measure its costs and show its benefits. In Section 4, we present two case studies of PLACE usage. We describe related work in Section 5, future directions in Section 6, and conclude in Section 7.

2 PLACE: Design and Implementation

In this section, we describe the PLACE system for controlling file layout. We first provide background, describe our goals in implementing PLACE, and then describe the API as exposed through the PLACE ICL. After presenting the programming interface, we discuss the PLACE implementation, including system initialization and the shadow structures it uses to control file placement. We also discuss general operation, issues of concurrency, and some limitations of the current implementation.

2.1 Background

Many modern UNIX file systems are based on the Berkeley Fast File System[13], including direct descendants found in the BSD and Solaris families, and intellectual descendants such as Linux ext2 [29]. One of the main innovations of FFS is the emphasis placed upon *locality*—by placing related data objects near one another on disk, FFS provided a quantum leap in performance over file systems that scattered data across the disk in an oblivious manner.

The primary construct used in FFS to manage disk locality is the *cylinder group* (or *block group* in ext2, terms that we will use interchangeably for simplicity). A cylinder group divides the disk into a number of contiguous regions, each of which consists of inodes, data blocks, bitmaps for tracking inode and block usage, and a small number of blocks that store implementation-specific information. By placing related data objects into a cylinder group, and conversely spreading unrelated objects across different groups, locality of access can be achieved.

The difficulty, of course, is deciding exactly which objects are “related” and which are not. Typically, simple heuristics based on the file system namespace are used. Specifically, to group related objects, most implementations place the inodes and data blocks of files within the same directory into the same group, assuming locality of access among those files. Conversely, new directories are placed in different groups, so as to spread presumably unrelated files across the disk (thus leaving some “room to grow” in each group). The original FFS implementation (and some descendants) spread large files across groups, so as to avoid filling one group with a single large file.

In designing the PLACE ICL, we seek to exploit our gray-box knowledge of how FFS-like systems perform file layout in order to allow users to better control where their files are placed on disk. We also wish to understand the limits of such gray-box control, including the types of functionality that cannot be realized on top of modern file systems.

2.2 Design Goals

In designing PLACE, we have the following goals:

- **Simple and intuitive control over layout:** Applications should be given a straightforward representation of disk locality, which they can then exploit with their own application-specific knowledge to improve I/O performance.
- **Easy to use:** PLACE should be as easy to use as possible – no substantial code modifications should be required. Both programming APIs and command-line tools should be provided.
- **Compatible with non-PLACE applications:** Applications that do *not* use PLACE on top of a given file system should operate as before, *i.e.*, basic file system structure and usage for unmodified applications should not change.
- **Unaffected file system namespace:** Applications (and users) should be able to name files according to whatever conventions they desire – layout should not be dependent upon specialized naming schemes.

As we will see below, these goals impact both the design and the implementation of PLACE.

2.3 Abstractions and API

As its basic abstraction, PLACE exposes the underlying *groups* of FFS-like file systems to applications that link with the PLACE library. Applications can then use knowledge of their own access patterns to place related files and directories into a specific group, thus exploiting spatial locality for improved performance.

The number of each group also provides applications with two other pieces of information. First, applications can safely assume that files in proximate groups are reasonably “close” to one another, *e.g.*, an object in group 1 is close to an object in group 2, but not likely to be very close to an object in group 55. Second, lower group numbers are located near the outer tracks of the disk, whereas higher group numbers are located near the inner tracks. Applications may wish to utilize zone-sensitive placement for large files and thus improve throughput.

Note that more abstract “virtual” groupings and even group hierarchies could be layered on top of the physical group interface if desired. However, for simplicity, we focus solely on this lowest level of abstraction in the rest of this paper.

To allow applications to place files and directories into specific groups, PLACE provides two basic functions to applications:

- `Place_CreateFile(char *pathname, mode_t mode, int group);` Creates a file specified by `pathname` and with mode set to `mode` in group number `group`. The first two arguments are identical to the `creat()` system call.
- `Place_CreateDir(char *pathname, mode_t mode, int group);` Creates a directory specified by `pathname` with mode set to `mode` in group number `group`. The first two arguments are identical to the `mkdir()` system call.

The `Place_CreateFile` call allows the fine-grained placement of files into particular groups, whereas the `Place_CreateDir` function allows applications to create a directory in a controlled manner. Subsequent file allocations in that directory (through PLACE or not) are then likely to be collocated, due to standard FFS policy.

Of course, PLACE may not be able to allocate the file or directory into a particular group, due to insufficient resources (*i.e.*, there are no free data blocks or inodes left in the group). In such a case, the standard behavior is for the routine to return an error indicating why and the object is not created. An alternative interface can be used in which the routines can instead search for a “nearby” group upon failure, and place the file or directory therein.

Several other utility and convenience functions are also provided. For example, applications can discover the number of groups in a given file system, the current utilization level of each group, and the number of the group that is currently the least utilized.

When a user does not wish to or cannot re-write an application to use the PLACE API, a set of command-line tools can be utilized instead. These tools allow users to move directories and files to specific groups, or to create them in specific groups; subsequent data access by unmodified applications will thus enjoy the benefits of the rearrangement.

2.4 Basic Operation

PLACE exploits the FFS tendency to use the file namespace as a hint for placement to gain control over file layout. To do so, PLACE must first create a structure in which new files and directories can be created in a controlled fashion; once created therein, the PLACE library then renames the files, moving them back into their proper location within the file system namespace. This file system structure, known as the *shadow directory tree* (SDT), is central to the PLACE implementation.

At initialization (a process that is performed once per new file system), PLACE produces an SDT structure that appears in the file system namespace as shown in Figure 2.

There are three important entities found within the SDT. First, the `.superblock` file contains persistent in-

```

/.hidden/.superblock
/.hidden/.concurrency
/.hidden/D1/
/.hidden/D2/
...
/.hidden/Dn/

```

Figure 2: **The Shadow Directory Tree.** The hidden shadow directory tree structure is presented. The `.superblock` file contains persistent information needed by PLACE, and the `.concurrency` file is used to manage multi-user access. Finally, the directories `D1` through `Dn` are used to control file placement.

formation about PLACE. Second, the `.concurrency` file is used to manage concurrent access to files through the PLACE API. Both of these files are discussed in more detail below. Third, and most interesting, is the set of directories named `D1` through `Dn`, where `n` is the number of cylinder groups in the file system. The initialization procedure (also described in more detail below) ensures that directory `Dk` is placed into cylinder group `k`. Note that all of these structures are placed in a “hidden” directory so that most applications will not see them when traversing the directory tree.

2.4.1 Controlling File Creation

With the SDT in place, creating a file in a particular group is straightforward. An application calls `Place_CreateFile`, passing in the pathname of the file to be created, the mode bits, and the desired group `k` within which to place the file. Internally, the PLACE ICL creates a file in the `Dk` shadow directory, and then simply calls `rename()` to put the file in the proper location in the namespace.

PLACE also checks to make sure that the file is allocated to the group the user requested, by looking up the `i`-number of the newly allocated file. During initialization (described below), PLACE learns of and records the `i`-number to group mapping, and uses that information here to determine if the allocation was successful.

2.4.2 Controlling Directory Creation

Placing a directory into the proper group with `Place_CreateDir` is more challenging; creating a directory in the proper `Dk` shadow directory does not suffice, as FFS-like file systems will place the child directory in a different cylinder group than its parent. Thus, a different approach is required, as shown in Algorithm 1.

The algorithm works by creating a temporary directory, checking if it is in the desired group (via its `i`-number), and

```

repeat
    tmp = PickNewName();
    mkdir(tmp);
    if (InDesiredGroup(tmp)) then
        break;
    end
    FillOtherGroups();
until forever;
rename(tmp, dirname);

```

Algorithm 1: **Directory Creation Algorithm.** The basic algorithm used to create a directory in a specific group on disk is presented.

repeating this process until the temporary directory is created in the correct group. When that directory is created, it is renamed to the proper location in the namespace.

One complication arises due to the particular directory allocation policies of some FFS-like file systems. For example, the Linux `ext2` policy searches for a group with an above-average number of free inodes and the fewest allocated data blocks, whereas NetBSD FFS picks the group with an above-average number of free inodes and the fewest allocated directories. Thus, the algorithm must be willing to create temporary files as well as directories to coerce the file system into creating a directory in the desired group. This process, referred to in Algorithm 1 as `FillOtherGroups()`, creates some number of files in each of the non-target groups. To ensure that the files are not spread across different groups in an uncontrolled manner, PLACE creates “small” files (*i.e.*, files that do not utilize any indirect pointers).

Unfortunately, this basic algorithm can be quite slow, as we will demonstrate in Section 3. To speed up the process in the common case, we build a *shadow cache* of directories with known group numbers within the SDT. Before attempting to create a new directory within a particular group, the directory creation algorithm first consults the shadow cache to see if a directory within that group already exists; if so, PLACE simply renames that directory and is finished, thus avoiding the expensive directory creation algorithm.

If PLACE does not find the appropriate directory in the cache, it performs the full-fledged algorithm as described above. In this case, the directories that are created during the algorithm can be added to the cache, thus repopulating the shadow cache periodically.

2.5 SDT Initialization

We now discuss the initialization process required by PLACE, as encapsulated within a tool we call `pmkfs` (for “PLACE mkfs”). There are two steps to `pmkfs`. First,

`pmkfs` discovers various system parameters which are used in the algorithms described above. Second, `pmkfs` creates the SDT on-disk data structures and populates the shadow cache.

2.5.1 Parameter Discovery

PLACE requires several pieces of information to create the on-disk structures to support controlled allocation. These are the number of groups in the file system (N_{grp}), and the number of blocks (B_{grp}) and inodes (I_{grp}) per group. The total number of blocks and inodes in the system are readily available via the `statfs()` system call.

Finding the number of groups is slightly more challenging. Our current algorithm calculates this number by allocating directories and recording the difference in the inode numbers of subsequently allocated directories. Since each directory is likely to be in a new group, the most common difference is the number of inodes per group. PLACE detects when allocation has “wrapped around” by the fact that a new directory will have an i-number that is quite close to a previously allocated directory (usually, one more). Once one knows I_{grp} , one can calculate the group number (G_{num}) of an object from its inode number (I_{num}) by computing: $G_{num} = (I_{num} - 1) / I_{grp}$.

The system also calculates the number of direct pointers used in an inode (*i.e.*, the size of a “small” file), which is required for the directory creation algorithm to work across multiple FFS platforms. This value is discovered by synchronously writing blocks into a file, and monitoring the number of free blocks in the file system via `statfs`. The “small” file size is discovered at the point where a single allocating block write decreases the free block count by two blocks, indicating that an indirect block has been allocated.

In the current implementation, PLACE requires exclusive access to an empty file system during initialization. The only reason for this restriction is that the value of I_{grp} is not exported by the file system and the procedure described previously must be used to determine it. If this number were made available from an outside source (*e.g.*, the system administrator), then PLACE could be initialized on top of a system already in use.

2.5.2 SDT Creation

In the second step, `pmkfs` stores the necessary information into the `.superblock` file, and then creates the directory tree containing directories D1 through Dn, assuming n groups. The process of creating these directories is identical to the directory creation algorithm found in Algorithm 1. As in the typical directory creation procedure, excess directories that are created are added to the shadow cache. In general, PLACE tries to maintain a minimal

threshold of shadow directories per group, so as to avoid the costly directory creation algorithm.

To obtain a better understanding of what this threshold should be, we examined file system traces from HP Labs [17]. During a typical busy day, we found that a few thousand long-lived directories were created, giving us a rough upper bound on the number of shadow directories PLACE would need to maintain to absorb a day’s worth of controlled directory creation in that environment.

2.6 Other Issues: Crash Recovery and Concurrency

During both file and directory creation, PLACE may create files and directories in the SDT, and thus there is the potential that data will accrue there over time; this will occur, for example, when a file is created in the SDT but the system crashes before the `rename` has taken place, or worse, if a job is killed in the midst of a PLACE library call. PLACE must thus include a basic crash recovery mechanism in order to periodically remove these files. We refer to this process as *SDT cleaning*.

Our current implementation of the SDT cleaner scans the SDT directory structures and removes any data objects that are “old” and thus left over from system crashes. As for how often to run the cleaner, many alternatives are possible. Our current implementation invokes the cleaner once every c invocations of PLACE (currently, c is set to 1000, which is probably too conservative), and whenever the longer directory-allocation process is run. Other alternatives include running the cleaner once per time interval (*e.g.*, once every day), or in a background process.

New issues also arise when considering PLACE usage under multiple processes or users. Concurrent use of PLACE by different processes is only a problem in the current implementation when using the basic algorithm to allocate a directory. In that situation, competing controlled directory creations in different groups could lead to significant difficulty in creating a directory in the desired location(s). To avoid this problem, PLACE acquires an advisory lock on the `.concurrency` file during this mode. This lock is only used to signify usage of the basic algorithm. In practice the usage of the basic algorithm repopulates the shadow cache, reducing the need for this mode of operation. A more cooperative approach is possible, where processes share the work of gaining control, but this would introduce significant complexity.

Multiple users also introduce a new issue, particularly as to whether the SDT should be shared or private per user. Sharing requires some level of trust among applications, as the SDT must be in a writable location. Thus, a shared SDT is vulnerable to many types of attacks (*e.g.*, changing the structures of PLACE to lead to poor allocations, or filling the SDT and causing a denial of service).

In many environments, this is not a problem, as a single user or application may have sole access to the file system. However, in less trustworthy settings, the SDT could be replicated on a per-user basis; although this increases space utilization and duplicates effort, it circumvents the security issues that arise due to sharing.

2.7 Limitations

The primary limitation of PLACE is that it is currently implemented only for FFS-like file systems. However, most modern UNIX file systems are FFS-like, and recent features, including journaling [29] within ext3 or the Soft Updates found within the BSD family of FFS implementations [23], do not affect our ability to control file placement with the techniques described above.

Another limitation arises due to the internal implementation of some FFS implementations, which spread larger files across different cylinder groups in order to avoid filling a single group too quickly [13]. This FFS behavior prevents PLACE from controlling where large files are laid out on disk, and thus we provide an interface to query PLACE as to the largest file size whose allocation can be “guaranteed” to be controllable. One notable exception to this standard FFS implementation strategy occurs within ext2, which does not spread larger files across different groups; this implementation strategy hints at what gray-box implementors would like to find inside of the systems they build on top of – behavior that is simple to understand and thus relatively easy to control.

PLACE also does not directly allow for fine-grained placement of files *within* a particular group. However, applications can modify the order of file creation to pack files into a group in a controlled fashion [1].

One alternative that we had initially explored overcomes these limitations but does not mesh well with applications that do not use PLACE. In this alternative approach, PLACE initially fills the target file system with a set of dummy files; by discovering the exact locations of each file, PLACE can then free up space whenever applications request new space, and thus *all* data allocations can be controlled. However, we deemed this approach unacceptable, as unmodified applications would not work correctly – to those applications, the file system appears as if it is full.

3 Analysis

In this section, we analyze the behavior of PLACE, demonstrating its functionality and its basic overheads. We first discuss the experimental environment, and then proceed through a series of microbenchmarks, demonstrating the effectiveness of layout control, and revealing

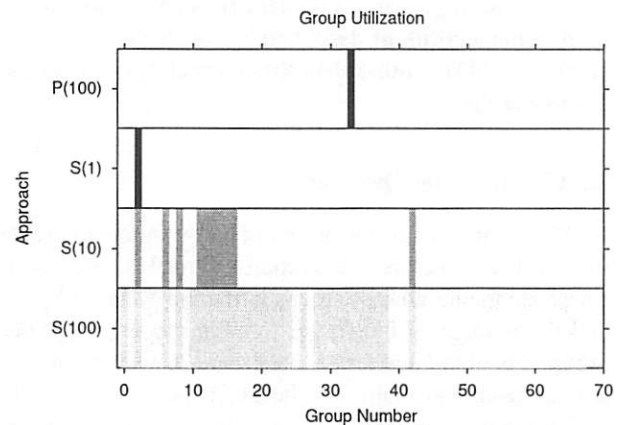


Figure 3: **Controlled Allocation.** The graph depicts four different experiments, each of which creates 250 200-KB files. In the first three, the standard file system interfaces are used, but the number of directories under which the files are created is varied, from 1 to 10 to 100; these three experiments are labeled *S(1)*, *S(10)*, and *S(100)*, respectively. In the fourth experiment, the PLACE API is used to create those files under 100 directories, but to place them in a single group in the middle of the disk (labeled *P(100)* in the graph). The group number is varied along the x-axis, and the shaded bar indicates some data has been placed in a particular group, with darker bars indicating more data. The `debugfs` command is used to gather the needed information.

the costs of system creation and usage. We then show how much improvement can be expected when reorganizing data and controlling layout to account for zoned-bit recording. Finally, we discuss our experience upon a broader range of OS platforms.

3.1 Experimental Environment

We present results with PLACE on top of the Linux 2.2 ext2 file system. All experiments on this platform are performed on a 550 MHz Pentium-III processor, 1 GB of main memory, and a 9 GB IBM 9LZX disk. The default ext2 file system built over this disk consists of 68 block groups. We also report on our experience with other file systems at the end of the section.

3.2 Layout Control

We begin with a simple experiment to demonstrate that PLACE can effectively collocate files into a specific group on the disk. Specifically, we compare four different methods of creating a 50 MB directory tree, allocated across 250 uniformly-sized files. In the first three, we use the standard file system interfaces, and alter the number of directories under which to place the files, from 1 to 10 to 100. In the fourth, we use the PLACE ICL to create the files underneath of 100 directories, but direct the system

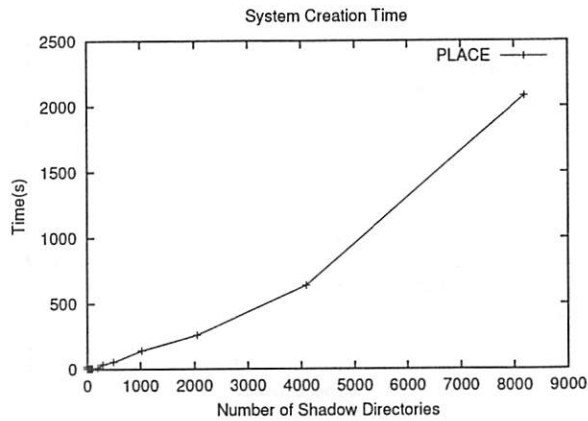


Figure 4: **System Initialization.** System initialization time is plotted. Along the x-axis, we vary the number of shadow directories created per group, and the y-axis plots the total time for the initialization to complete.

to place the files and directories into a single group in the middle of the disk. Figure 3 shows the group utilization of each approach for directory trees of 50 MB of data.

As we can see from the figure, with more directories and the standard layout algorithms, the data from the files is scattered across the disk. In contrast, with PLACE, all of the data is located in the middle group of the file system, exactly as desired.

3.3 System Creation

Now that we have demonstrated basic control over layout, we seek to understand the costs of using the system. The first cost that we present is that of system initialization, as performed by the `pmkfs` tool. Figure 4 presents system initialization time.

The dominant cost of system initialization is in the number of shadow directories that are created within the shadow cache. Therefore, we present the sensitivity of initialization time to the number of shadow directories created per group. As one can see from the figure, this cost does not scale well with an increasing number of directories in the Linux ext2 system, as an increasing amount of data needs to be created in order to allocate directories across all of the groups successfully.

3.4 API Overheads

We next present the overheads of controlled file and directory creation via PLACE. Our goal here is to understand the costs of gray-box control over data placement. Table 1 breaks down the cost of creating different-sized files through the `Place.CreateFile` interface.

The costs presented in the table are broken down into

	Time (ms and Percentage)			
	0 B	8 KB	64 KB	1 MB
Base	0.12 7.7%	0.20 12.0%	0.79 34.5%	9.80 85.6%
State	1.19 75.5%	1.20 70.8%	1.20 52.3%	1.23 10.7%
Alloc	0.14 9.2%	0.15 8.7%	0.15 6.4%	0.15 1.3%
Ren	0.04 2.6%	0.04 2.5%	0.04 1.9%	0.08 0.7%
Misc	0.08 5.0%	0.11 6.4%	0.11 4.9%	0.20 1.7%
Total	1.57 ms	1.69 ms	2.29 ms	11.45 ms

Table 1: **File Allocation Overheads.** Each result shows the average of 100 controlled file creations using the PLACE ICL. There was little variance (less than 0.04 ms) across the runs.

	Shadow Cache	Time (ms and Percentage)			
		Without Shadow Cache			
		Min	Median	Max	
Base	0.08 4.4%	0.10 3.0%	0.10 0.4%	0.00* 0.0%	
State	1.17 63.4%	1.47 46.4%	1.19 4.8%	0.00* 0.0%	
Alloc	0.24 12.7%	0.99 31.4%	5.46 22.1%	3.29 69.9%	
Ren	0.04 2.0%	0.03 1.0%	0.03 0.1%	0.00* 0.0%	
Clean	N/A	0.22 7.2%	17.5 70.7%	1.41 30.0%	
Misc	0.32 17.5%	0.35 11.0%	0.47 1.9%	0.01 0.1%	
Total	1.85 ms	3.16 ms	24.7 ms	4.71 s	

Table 2: **Directory Allocation Overheads.** Each result shows the average of 100 controlled directory creations using the PLACE ICL. Note that while most times are in milliseconds, the rightmost column (Max) shows time in seconds. The '*' indicates that the time shown is not actually zero, but appears as such due to rounding.

five different categories, across four different file creation tests. The five categories are as follows: **Base**, the time to create the file itself through standard interfaces; **State**, the time to read the `.superblock` file to access system statistics and configuration information; **Alloc**, the time to control allocation (in this case, a `stat` system call to check the inode number); **Ren**, the time to rename the file into the correct namespace; and **Misc**, additional software processing overhead.

As we can see from the table, the PLACE API for file creation adds roughly a 1 ms overhead to file creation. This cost is mostly due to PLACE initialization, which would be amortized over multiple calls to the PLACE library. However, there is still significant overhead in the allocation, rename, and other software overheads. Finally, as file size increases, the overheads are also (unsurprisingly) amortized.

We next explore the overheads of directory creation via the `Place.CreateDir` API. Table 2 presents the cost breakdown of a controlled directory allocation, both with and without the shadow cache. Note that a new category is also included, labeled **Cleanup**, which includes time spent cleaning up the SDT after the directory-allocation process has run. Also note that **Alloc** in this case refers to

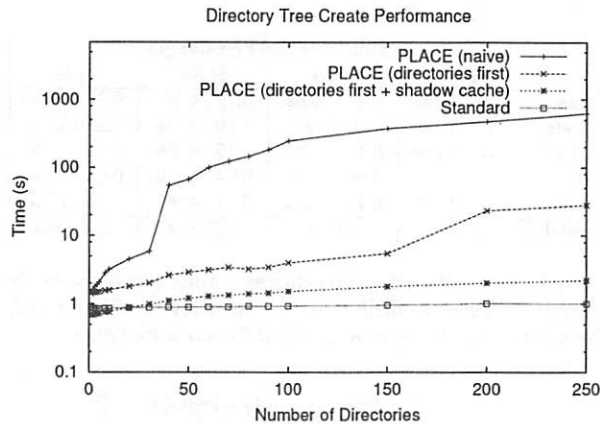


Figure 5: **Create Performance.** The cost of moving a directory tree into a specific group is presented, varying the number of sub-directories in the structure along the x-axis, given a fixed amount of data (50 MB, spread evenly across 250 files). Four different approaches to creating the structure are compared, as described in the text. The y-axis presents the total time for the bulk collocation, on a log scale.

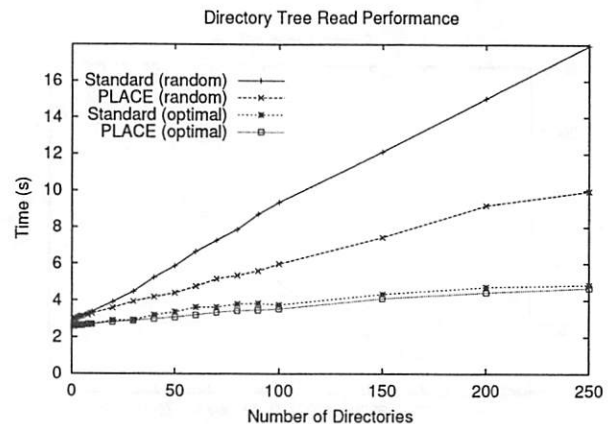


Figure 6: **Small-file Reads.** The time to read 250 200-KB files (50 MB) of data is shown, under four different settings, varying the number of directories across which the data is placed. In two settings, the standard file system APIs are used to create the files. In the other two settings, PLACE is used to collocate all data into a single group. Two orders are shown: 'random', which reads the files in random order, and 'optimal', which reads them in a single scan of the disk.

the costs of creating any necessary files or directories as required by the directory-allocation algorithm.

From the table, we can make a number of observations. First, with the shadow cache, the time for a controlled directory creation is reasonable, at roughly 1.85 ms (however, this value is still substantially higher than the base directory creation cost, which is approximately a factor of 20 faster). Second, without the shadow cache, times are higher, with a median cost of around 25 ms. The column that lists the maximum time without the shadow cache indicates the potential cost of running the full directory-creation process; in the worst case, it takes over 4.7 seconds to create a directory in the correct group. This difficulty arises with a controlled creation within the last (and hence smaller) group; because ext2 allocates directories based on free bytes remaining, it takes an excessively long time to fill up the other groups and hence coerce a directory allocation into this last group. The **Base**, **Alloc**, and **State** times are essentially constant, and constitute a negligible part of the total in the maximum case. Since the shadow cache does not use the basic algorithm, it does not need to **Clean** afterward.

3.5 Bulk Collocation Costs

A common usage of PLACE is to move an entire directory tree into a specific group on the disk, which can be accomplished with one of the PLACE command-line tools. Thus, we were interested in what strategy this tool should take in moving a large amount of data from the source to its final destination within one group (or a small number of groups).

Figure 5 presents the time to perform this “bulk collocation” of 50 MB of data, again spread evenly across 250 200-KB files, under a varying number of sub-directories. Four schemes are compared. The first uses PLACE in a naive fashion, by creating directories and files in the target group recursively, and assuming that no shadow cache exists. This approach is dramatically slow, as the directory creation algorithm finds it increasingly difficult to force data into the target group. The second approach creates directories first, and performance improves tremendously, because the ext2 allocation policy uses the number of bytes allocated in its group-selection policy. Thus, by not creating files in the target group, it is much easier to coerce the system into choosing it. The third scheme shows the time for the second approach assuming that directories can be allocated from the shadow cache, which also improves the performance of the bulk collocation down to just a few seconds. Finally, a traditional directory-tree copy is shown as a comparison point; it is fast because it does not have any overhead associated with it, even though it is likely to spread data across the disk in a less localized manner.

3.6 Benefits of Collocation

To quantify the potential read performance improvement of PLACE, we perform a final set of microbenchmarks. Figure 6 shows the performance of the first set of tests, which present the time it takes to read a set of 250 200-KB files that have been collocated on the disk.

From the figure, we can see that if an application reads a

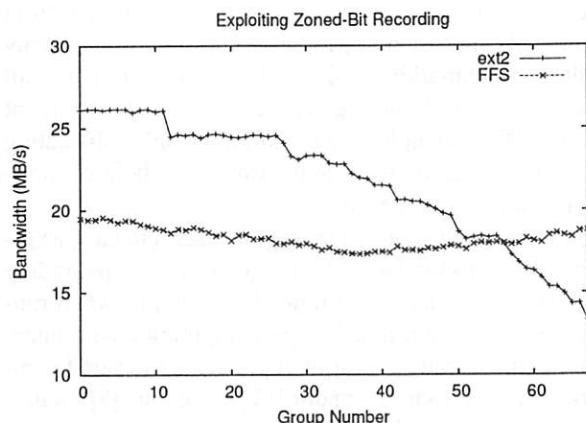


Figure 7: **Large-file Reads.** The performance of reading a 100-MB file is shown on both ext2 and FFS, while varying the group in which the file is created along the x-axis. The file cache is flushed via a umount/mount cycle before the read to ensure that the disk bandwidth is properly measured. Each point is the average of three trials; the variance across all trials was low.

set of files in random order, collocating them into a localized portion of the disk improves performance by almost a factor of two (the 'random' lines in the graph). However, if the files are read in the optimal order (essentially just scanning across the disk in a single sweep), the benefits of collocation are quite small; in this case, spreading data across the disk results in only a few additional seeks, and thus makes little overall difference in performance.

We also demonstrate how PLACE can be used to take advantage of the zoned bandwidth characteristics of modern disks [15]. Figure 7 plots the performance of large sequential file scans, when the files are placed into specific groups. The figure depicts the performance of PLACE on two file systems: standard ext2, and a modified ext2 which acts like traditional BSD FFS.

As one can see from the figure, on the ext2 platform, placing data in the lower-numbered groups corresponds directly to placing data onto the outer zones of the disk, thus improving performance. We also observe that when the same experiment is run on top of an "FFS" file system, the zoning nature of the disks are hidden; because FFS spreads blocks of large files across the disk, PLACE cannot control the placement of those blocks.

3.7 Experience with Other Systems

Our primary focus has been on the ext2 file system, as it is a modern implementation of FFS concepts and a popular file system in the Linux community. However, we designed many aspects of PLACE with more general FFS-like systems in mind; therefore, we were interested in studying the behavior of PLACE on other platforms.

Our first test of generality was to run PLACE on top of an ext3 file system, the journaling version of ext2 [29]. Because ext3 goes to great lengths to preserve backwards-compatibility with ext2, the same on-disk structures are utilized. Thus, we were not surprised to find that PLACE works without issue on top of ext3.

We also tested PLACE on top of an implementation of the FFS [13] allocation algorithms in the Linux kernel. PLACE worked without modification in this environment, with the limitations discussed in Section 2.7 and shown directly in Figure 7 relating to the placement of large files.

4 Case Studies

In this section, we describe two different example uses of the PLACE library. In the first, we demonstrate how a web server can reorganize files with PLACE so as to improve server throughput and response time. In the second, we describe how a high-speed file system benchmarking infrastructure can use PLACE to quickly extract I/O characteristics from the underlying system.

4.1 Improving Web Server Throughput

In our first example, we apply PLACE in order to understand the potential performance improvement in a web server environment. By reorganizing files such that the most popularly accessed files are close to one another on disk, seek costs can be reduced. Web service is a particularly good target for PLACE, as the structure of a typical web directory tree does not necessarily match the locality assumptions encoded into most FFS-like file systems. Further, there is no need to change the source code of the web server; the reorganization can be performed off-line via command-line tools.

We study the potential benefits through a simplified trace-based approach. We utilize a web trace from the University of Wisconsin-Madison web server. The trace is first preprocessed to remove requests that do not induce file system activity, such as errors and redirects, and the only requests that remain are ones that transfer data and HTTP 304 replies (a reply to a cache coherence check). The trace contains roughly 2.6 million requests, and accesses a total directory tree size of 720 MB.

To understand the potential gains of collocation, we run the trace through a file system request generator. For each trace entry, the request generator invokes the appropriate file system call and records the response time. Specifically, to model HTTP 304 requests, the generator performs a `stat` system call on the file, and for requests that transfer data, it maps the file into memory and touches every page. Although this approach does not capture the full complexity of a web environment, it should give us a

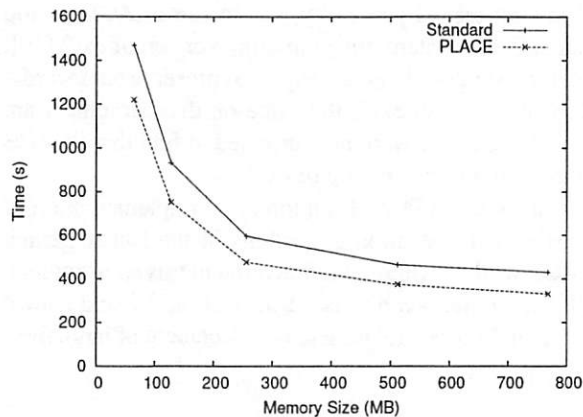


Figure 8: **Web Server Performance.** The time to play back the I/O component of a web trace is shown. The standard line plots the performance with typical layout, and the PLACE approach packs all data into a small portion of the disk. Each point represents the mean of three trials (the variance was low), and the size of the directory tree being served is 720 MB.

baseline for the potential performance improvement from file system reorganization.

We utilize the PLACE command-line tools to collocate the directory tree into the outer-most tracks of the disk, and compare this organization to a directory tree spread across the drive as determined by typical file system heuristics.

Figure 8 gives the time to replay the trace as a function of the amount of memory. Across the range of memory sizes, collocation with PLACE improves performance by roughly 20%. These benefits result directly from a reduction in seek costs, as demonstrated by further instrumentation of our testing apparatus. Specifically, by recording the group number of each file access, we can compute the average *group distance* traversed between requests. For the standard layout, the average number of groups between requests was 6.02, while for PLACE it was 0.52.

However, overall performance gains were limited due to the access patterns found in this particular web trace. In the trace, 76% of the requests were for 122 files in a single image directory. These files are thus collocated under standard FFS policy, reducing the need for PLACE-assisted file placement. However, even in such an environment, PLACE was able to improve performance in a simple and direct manner; a greater benefit can be expected in environments where access patterns do not match directory structure so closely.

4.2 Rapid File System Microbenchmarking

In our second example, we examine the use of PLACE in a different context, that of fast discovery of I/O performance characteristics. Many tools have been developed

over time that extract performance characteristics from the underlying system [5, 14, 20, 21, 28]. However, many of these benchmarking tools need to be run as root, and all run for an uncontrolled (and potentially lengthy) amount of time. For example, Chen and Patterson's self-scaling benchmark runs for many hours (even days) before reporting results back to the user [5].

In some settings, it would be quite useful to have a system benchmarking tool that ran quickly, perhaps trading accuracy for a shorter run-time. For example, when running an application in a foreign computing environment (e.g., Seti@Home [27], or in any wide-area shared computing system such as Condor [12] or Globus [8]), a mobile application needs to quickly extract the characteristics of the underlying system so that it can parameterize itself properly to the system. Further, the benchmark must be run entirely at user-level, requiring no special privileges to discover system parameters.

Thus, we develop the benchmarking tool FAST (Fast or Accurate System exTraction), that allows a mobile application to extract various performance characteristics from the underlying system under a fixed time budget and entirely at user level. Although FAST currently can extract information about both the I/O system and the memory system, only the I/O system component utilizes PLACE.

As an example of a mobile application, we examine the single processor version of NOW-Sort, a world-record-breaking sorting application [2]. While traditionally thought of in database contexts [16], sorting is also commonly found in many scientific computation pipelines [10], and therefore it is a reasonable candidate for mobile execution in scientific peer-to-peer shared computing systems [8].

NOW-Sort requires three parameters to tune itself to the host system. The first two are I/O parameters: the bandwidth expected from the local disk, and the worst-case seek time. With these two numbers, the sort can estimate how large its buffers must be during the merge phase in order to amortize seek costs. The third is the size of the caches in the memory-hierarchy. By sorting data in cache-sized chunks, sorting proceeds at a much faster rate [16].

The most difficult of these parameters to generally extract is the maximum seek cost. However, with the PLACE API, the FAST tool can create two files that are far apart on the disk, issue a synchronous update to the first, start a timer, issue a synchronous update to the second, and record the elapsed time of the second write, giving a coarse estimate of a full-stroke seek. Further refinements can be made over time, in order to remove rotational costs if so desired.

Table 3 presents the costs of running FAST on our test system. In this mode of operation, FAST runs as quickly as possible, garnering coarse estimates of the required system parameters. From the table, we observe that the

	Time (s)
Cache	0.73
Bandwidth	2.46
Max Seek	0.52
pmkfs	4.51
Total	8.22

Table 3: **FAST Performance.** The table presents the time FAST takes to discover system parameters. In this mode, FAST is configured to run as quickly as possible, extracting coarse estimates but consuming less overall time.

total time to extract the needed information for sorting is roughly 8 seconds. For sorts of massive data sets, spending the extra few seconds to configure the application is well worth the time. Finally, note that `pmkfs` is specialized to the task at hand; by giving it command-line options so as to prevent the creation of any shadow directories, initialization time is reduced to a small, fixed overhead.

5 Related Work

The work most directly related to PLACE is the gray-box File Layout Detector and Controller (FLDC) described in the original gray-box paper [1]. The FLDC has two components: the first can be used to decide in which order to access a set of files, and the second to re-write file within a particular directory so as to likely improve later accesses. Both components could be useful here; however, PLACE goes well beyond FLDC, exposing fine-grained control over file and directory layout to applications.

Applications have long sought better control over underlying operating system policies and mechanisms [26]. In response to this demand, previous research has developed new operating systems, including Spin [3], Exokernel [7], and VINO [22], that allow much-improved control over operating system behavior. The gray-box approach provides a different route to improved control over the underlying OS; by exploiting knowledge of OS behavior, PLACE demonstrates that file and directory layout can be realized at user-level.

The PLACE method of exposing group numbers is conceptually similar to the Exokernel philosophy of exposing physical names. PLACE treats the file system as the underlying entity and exposes its internal structure (e.g., ext2 block groups), while Exokernel exposes the details of the hardware (e.g., physical sector numbers).

Moving data blocks into a better spatial arrangement, as we do in the web server case study, has been explored in many other contexts. For example, in their work on disk shuffling, Ruemmler and Wilkes track fre-

quency of block accesses, and reorder disk blocks to reduce seek times [19]. At a higher level, Staelin and Garcia-Molina rearrange where files are placed within the file system [25]. The major difference between these approaches and PLACE is that they are performed transparently to users and applications; no control is exposed. However, both are more sophisticated in tracking which blocks or files are accessed in temporal succession; we hope to develop an access-tracking tool in the future.

Our work on improving web server performance is similar to other work on improving web proxy performance. For example, Hummingbird is library-based file system designed for web proxies [24]. PLACE provides some of the same features of Hummingbird in that it allows the users to collocate files and does not tie locality to naming. In contrast, PLACE is implemented on top of an FFS-like file system, whereas Hummingbird performs its functions in a library that runs on a raw disk. Further, Hummingbird is specialized for a web-proxy environment, whereas PLACE is a general-purpose tool.

Finally, the FAST tool bears some similarity to recent work in database management systems. For example, in *online aggregation* [9], the DBMS returns an approximate result of a selection query to the user immediately, and includes a statistical estimate of the accuracy of the result. If the user allows the query to keep running, the system refines the result over time, and as more data is sampled, the answer becomes more precise. The FAST tool applies this same philosophy to a benchmarking system.

6 Future Work

A number of avenues exist for future research. First, we plan to explore the breadth of applicability of PLACE on top of other file systems. One platform we are interested in is the BSD family; we believe there are some new challenges in this domain, as more recent BSD implementations of FFS utilize the DirPrefs algorithm for directory group selection [6]. This algorithm places directories near their parents, in an attempt to increase the performance of certain common operations (e.g., the unpacking of a large directory tree). Building a gray-box controller such as PLACE on top of DirPrefs would thus require that extra care be taken to spread directories across groups.

Building PLACE on top of a log-structured file system (LFS) [18] would also be interesting. For example, grouping of particular related files would generally be straightforward; if a user wishes to group two files, those files can be written out at the same time. However, other aspects make LFS more challenging, including grouping of files that span multiple segments and controlling the off-line behavior of the cleaner. More generally, we are interested in developing techniques that can be used to control allo-

cation across a broader range of file systems.

We would also like to investigate the utility of these methods on a range of different storage devices. Specifically, we would like to determine how useful such controlled file placement is on top of modern disk arrays.

Finally, a tool such as PLACE is a low-level mechanism for placing files in a controlled manner on disk; exactly which files should be placed together is a higher-level policy decision that requires detailed knowledge of how files are accessed over time. Thus, similar to previous work in data rearrangement [19], we plan to develop a tool to track how files and blocks are accessed and thus generate the necessary inputs for better file placement.

7 Conclusions

In the classic paper *Hints for Computer System Design* [11], Lampson tells us: "Don't hide power." Higher-level abstractions should be used to hide the *undesirable* properties; useful functionality, in contrast, should be exposed to the client. Many UNIX file systems do not expose explicit controls for laying out files according to user demands. Given standard layout heuristics, workloads that do not conform to the locality assumptions set in stone nearly 20 years ago perform poorly.

In this paper, we present the design, implementation, and evaluation of PLACE, a gray-box Information and Control Layer that exposes file and directory information to applications. By exploiting knowledge of internal algorithms that are common to FFS-like file systems, PLACE can control file and directory allocations.

Through microbenchmarks, we have shown that the costs of gray-box control are not overly burdensome, and that the potential benefits of controlled allocation are substantial. Through two case studies, we have demonstrated that the PLACE system can be used in realistic and diverse application settings. We have also discussed the limitations of PLACE as well as the gray-box approach to controlled allocation, highlighting the features of file system allocation policies that make it simple or difficult to build control on top of them.

The gray-box approach provides an alternative path for innovation. Instead of requiring changes to the underlying operating system, which may be difficult to implement, maintain, and distribute, a gray-box ICL embeds some knowledge of the underlying system, and exploits that knowledge to implement new functionality, often in a portable manner. One important question remains: what is the full range of functionality that can be implemented in the gray-box manner, and what are the ultimate limitations? With each ICL, we take another small step toward the final answer.

Acknowledgments

We would like to thank Nathan Burnett, Tim Denehy, Florentina Popovici, Vijayan Prabhakaran, and Muthian Sivathanu for their feedback on this paper. A special thanks to Muthian for his assistance with the HP traces. We also thank Geoffrey Kuenning for his excellent shepherding, and the anonymous reviewers for their thoughtful suggestions, which in combination have greatly improved the content of this paper. We thank John Heim and the staff of DoIt for providing a recent web trace that included path names, and Tom Engle for the implementation of the FFS allocation algorithm in the Linux kernel. Finally, we thank the Computer Systems Lab for providing a superb environment for computer science research.

This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, an IBM Faculty Award, and the Wisconsin Alumni Research Foundation.

References

- [1] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.
- [2] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD Conference on the Management of Data (SIGMOD '97)*, Tucson, Arizona, May 1997.
- [3] B. N. Bershad, S. Savage, E. G. S. Przemyslaw Pardyak, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
- [4] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 29–44, Monterey, CA, June 2002.
- [5] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation—Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1992 ACM SIGMETRICS Conference*, pages 1–12, May 1993.
- [6] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD. In *Proceedings of the USENIX Annual Technical Conference (FREEINX Track)*, Monterey, California, June 2002.
- [7] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for

- Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
- [8] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.
 - [9] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD International Conference on Management of Data (SIGMOD '97)*, pages 171–182, Tucson, Arizona, May 1997.
 - [10] K. Holtman. CMS data grid system overview and requirements. CMS Note 2001/037, CERN, July 2001.
 - [11] B. W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 33–48, Bretton Woods, NH, December 1983. ACM.
 - [12] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of ACM Computer Network Performance Symposium*, pages 104–111, June 1988.
 - [13] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
 - [14] L. McVoy and C. Staelin. Imbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Winter Technical Conference*, January 1996.
 - [15] R. V. Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, January 1997.
 - [16] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *1994 ACM SIGMOD Conference*, May 1994.
 - [17] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, pages 14–29, Monterey, California, January 2002.
 - [18] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
 - [19] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-91-156, Hewlett Packard Laboratories, Oct 1991.
 - [20] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995.
 - [21] J. Schindler and G. R. Ganger. Automated Disk Drive Characterization. Technical Report CMU-CS-99-176, Carnegie Mellon, 1999.
 - [22] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.
 - [23] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 71–84, San Diego, CA, June 2000.
 - [24] E. Shriver, E. Gabber, L. Huang, and C. A. Stein. Storage Management for Web Proxies. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX '01)*, pages 203–216, Boston, Massachusetts, June 2001.
 - [25] C. Staelin and H. Garcia-Molina. Smart Filesystems. In *Proceedings of the 1991 USENIX Winter Technical Conference*, Dallas, Texas, January 1991.
 - [26] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.
 - [27] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A New Major SETI Project based on Project Serendip Data and 100,000 Personal Computers. In *Proceedings of the 5th International Conference on Bioastronomy*, 1997.
 - [28] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.
 - [29] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

Operating System I/O Speculation: How two invocations are faster than one

Keir Fraser, *University of Cambridge Computer Laboratory**

Fay Chang, *Google Inc.**

Keir.Fraser@cl.cam.ac.uk Fay@google.com

Abstract

We present an in-kernel disk prefetcher which uses speculative execution to determine what data an application is likely to require in the near future. By placing our design within the operating system, we provide several benefits compared to the previous application-level design. Not only is our system easier to implement and deploy, but by handling page faults as well as traditional file-access methods we are able to apply speculative execution to swapping applications, which often spend the majority of their execution time fetching non-resident pages. We also present two new OS features that further improve the performance of speculative execution for applications that have large page tables and working sets. These are a fast method for synchronizing an errant speculative process with normal execution, and a modified form of copy-on-write which preserves application semantics without delaying normal execution. Finally, by leveraging OS knowledge about memory usage and contention, we design a mechanism for estimating and limiting the memory overhead of speculative executions.

Our implementation for Linux 2.4.8 provides benefits of up to 60% for a wide range of explicit-I/O and swapping applications. Our results show that our support mechanisms for swapping applications provide significant performance benefits, and in some cases prevent speculative execution from hurting performance. We further demonstrate that our memory control mechanism effectively limits speculative overheads while allowing beneficial speculative executions to proceed unhindered.

1 Introduction

In the past decade, the gap between processing speeds and disk access times has increased by an order of magnitude [1]. At the same time, although memory sizes

have increased substantially, so have application data requirements. Systems therefore continue to swap their data sets to and from disk as they are often too large to all fit in memory.

In recognition of this problem, there has been a great deal of research into automating disk prefetching algorithms that are dramatically more accurate than the standard heuristics in current operating systems. Recent work by Demke and Mowry [2] demonstrated impressive performance results using compiler-based techniques. However, system-wide use of their approach would require recompiling every application. Moreover, their compiler analyses are limited to looping array codes.

To address this problem, Chang and Gibson [3] proposed to discover the future data accesses of an application stalled on I/O by executing ahead in its code, ignoring non-memory-resident data. They also proposed a design for automatically modifying application binaries to apply this *speculative execution approach*. With this design, they demonstrated that speculative execution can deliver substantial performance benefits for a diverse set of applications that issue explicit file read calls.

Unfortunately, their user-level design has several major disadvantages. First, their design relies on the correct implementation of a complex binary modification tool. Second, while their design does not require application source code, applications must still be transformed to receive benefit. Third, their design can only issue prefetches for disk reads caused by explicit file I/O because page faults cannot be trapped by the application. The system therefore cannot be applied to *swapping* (or *out-of-core*) applications which leverage file- and swap-backed virtual memory to avoid the complexity of explicit I/O. Finally, their design does not measure or limit the memory used by speculative execution, and its resulting effect on system performance. It is therefore poorly suited for use on realistic systems, which do not always have abundant memory.

In this paper, we demonstrate how these problems can be overcome with an *in-kernel* design for automating speculative execution. We present a design that,

* This work was performed while the authors were at Compaq Systems Research Center.

by leveraging existing operating system features, is not only substantially easier to implement but also may provide benefit to arbitrary unmodified application binaries. Moreover, by exploiting knowledge that is typically unavailable outside the operating system, our design automates disk prefetching for virtual memory accesses as well as explicit I/O calls, enabling it to provide benefit regardless of the I/O-access methods used by applications. In addition, we propose two new operating system features which substantially improve the performance of speculative execution for swapping applications. Finally, we describe a mechanism for estimating the impact of memory use by speculative executions on system performance, and thereby controlling speculative execution when memory resources are not abundant.

We have implemented our design within the Linux 2.4.8 kernel and evaluated it using eight applications, which include four explicit-I/O applications, three swapping applications, and one application that performs a substantial amount of both forms of I/O. We demonstrate that our basic design provides similar benefit to the prior design on explicit-I/O applications, while requiring much less implementation effort. We then demonstrate that our design also provides benefit for swapping applications, particularly with the assistance of the new features we identified. Finally, by varying the amount of usable memory on our system, we demonstrate the benefit of our mechanism for controlling the memory usage of speculative execution.

The remainder of this paper is organized as follows. Section 2 describes the speculative execution approach to automating I/O prefetching. Section 3 describes the testbed and benchmarks that we use to evaluate our proposals throughout this paper. Section 4 presents and evaluates the baseline version of our new in-kernel design and implementation. Then, in Sections 5 and 6, we proceed to describe our improvements for swapping applications, and our mechanism for controlling memory overhead. Section 7 contrasts our in-kernel design with the prior user-level design. Finally, Sections 8 and 9 discuss related work, future work, and conclusions.

2 Speculative execution

The *speculative execution approach* exploits the increasing abundance of spare processing cycles to automate prefetching for applications that stall on disk I/O. Usually, when an application needs some data that is not in memory, it will issue a disk request and then stall waiting for that request to complete. Rather than simply wasting unused processing cycles while applications are stalled on I/O, the speculative execution approach uses these cycles to try to discover and initiate prefetching

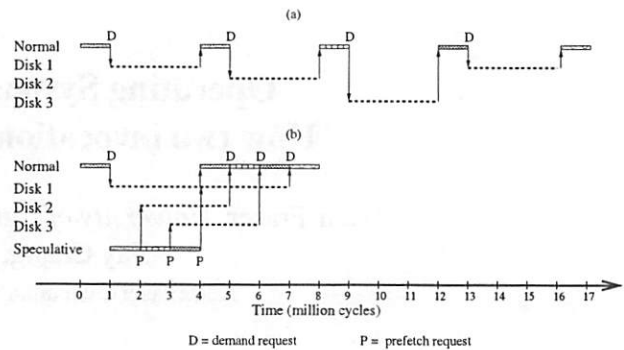


Fig. 1: Example illustrating how the speculative execution approach could reduce I/O stall time. (A) shows how execution would ordinarily proceed for a hypothetical application. (B) shows how execution might proceed for the application with the speculative execution approach. While normal execution is stalled on its first I/O request, speculative execution may be able to initiate prefetching for all the non-resident data that the application will access in the future. This could halve the application's execution time.

for the future data needs of stalled applications by running ahead of their stalled executions. In particular, the approach assumes that this speculative pre-execution of the application's code will be sufficiently similar to the application's future normal (non-speculative) execution that it will encounter the same accesses to non-resident data. Based on this assumption, speculative execution attempts to improve the application's subsequent performance by converting any such accesses to prefetches.

Figure 1 illustrates how this approach could deliver substantial performance improvements for a hypothetical application that accesses four non-resident data pages spread across three disks. For simplicity, assume that the application executes for one million cycles between each such access, and that a disk can service a request in three million cycles. When this application is executed, its execution will ordinarily alternate between processing and stalling on I/O. If the speculative execution approach were applied then, when normal execution stalls on its first I/O request, execution would continue speculatively. Whenever speculative execution encounters an access to non-resident data, it will instead issue a non-blocking prefetch call. In this manner, it may be able to initiate prefetching for all of the application's subsequent data accesses. When the original disk request completes, normal execution will resume. Now, however, its subsequent data accesses will be serviced out of main memory, halving the application's execution time.

It is worth noting that speculative execution will not be effective in all cases. For example, it will offer no benefit on systems where CPU, memory or disk are al-

ready fully utilized. Also, a speculative process will incorrectly predict future accesses if they depend on non-resident data. However, our success in applying speculative execution to a wide range of benchmark applications indicates that independent I/O accesses are common.

3 Experimental setup

Throughout this paper, we evaluate each successive design proposal after describing that proposal in order to isolate its performance impact and motivate further refinements. We present our experimental setup in this section to assist this progressive unfolding of our design.

We evaluate our design proposals on an 866MHz Pentium III configured to use 64MB of memory, running our modified Linux kernel. Most modern desktop computers have much more memory. We restricted the memory size deliberately to facilitate comparison with prior work [4, 3] and compensate for established I/O benchmarks [5] which use data sets that have not been updated to reflect growth in data set sizes. Our storage system consists of seven Compaq RZ1CB Ultra SCSI disks (12ms average access time). The file system is striped across four of the disks with a 64KB stripe unit. The central cylinders of the other three disks are designated as swap space. The maximum transfer rate supported by the disks and the SCSI interface is 40MB/s.

All our results are averages over three runs; however, the variance in execution time across these runs was always small (within a few percent of the calculated mean). All file- and swap-backed pages were flushed from memory before each run.

3.1 Benchmark applications

We use eight benchmark applications in our evaluation. Four of the benchmark applications are explicit-I/O applications, three are swapping applications, and one performs a substantial amount of both types of I/O. To assist comparison with prior work, these benchmarks are similar, and in many cases identical, to those used in prior evaluations of the TIP prefetching and caching manager [5], user-level speculative execution [3] and compiler-based prefetching [4, 2]. The benchmarks are summarized in Table 1 and described in greater detail below.

Agrep (version 2.0.4) is a fast pattern matching utility. *Agrep* opens each file on its command line in turn, and reads each one sequentially from start to finish. In the benchmark, *Agrep* searches 8971 files in the Linux 2.4.5 source tree for exact matches of a simple string that does not occur in any of the files.

Gnuld (version 2.11.2) is the Free Software Foundation's object code linker. *Gnuld* first reads each object file's file header and uses it to find the symbol header, which in turn provides offsets to the symbol and string tables. *Gnuld* then makes a small number of small, non-sequential reads to gather debugging information; the required file offsets are determined from the symbol tables. Finally, *Gnuld* sequentially reads the non-debugging sections in each object file. In the benchmark, an Alpha cross-linker is used to link 562 object files to produce a Digital UNIX kernel.

PostgreSQL (version 7.0.3) is an enhanced version of the original POSTGRES database management system. Our tests use a subset of the open-source database benchmark (OSDB), which implements the industry-standard AS3AP benchmark suite [6]. Our data set consists of four relations conforming to the AS3AP specification, which reside in 500MB of disk space. Our benchmark generates a set of indexes for each relation.

XDataSlice (version 2.2) is a data visualization package that allows users to view a false-color representation of arbitrary slices through a three-dimensional data set. The original application limited itself to data sets that fit into memory, but our version was modified for the TIP benchmark suite [5] to load data dynamically from large data sets. In the benchmark, *XDataSlice* retrieves 25 random slices through a set of 512^3 32-bit values which resides in 512MB of disk space.

FFTPDE and *MGRID* are two applications from the NAS Parallel benchmark suite [7], which have been modified by Demke and Mowry [2] so that their data sets do not fit in main memory. These applications make looping array accesses whose stride length and bounds vary dynamically during execution. This lack of predictability makes it hard for a conventional prefetcher to prevent I/O stalls.

MATVEC is a matrix-vector multiplication kernel that we obtained directly from Demke and Mowry's recent paper on compiler-based prefetching and memory management [2].

Finally, *Sphinx* is a speech recognition application. As with *XDataSlice*, the original application was modified to load its data dynamically from disk for the TIP benchmark suite [5]. The benchmark is to recognize an 18-second recording that was commonly used in *Sphinx* regression testing. The benchmark contains two phases: a first phase in which it reads about 15MB in a mostly sequential fashion from four data files, and a second phase of seemingly random accesses to a 176MB file.

Benchmark	Description	Run time	Read calls	Data set size
<i>Agrep</i>	text search	42 s	9385	105 MB
<i>Gnuld</i>	object code linker	30 s	16261	70 MB
<i>PostgreSQL</i>	AS3AP benchmark (database queries)	9915 s	4845047	535 MB
<i>XDataSlice</i>	visualization of 3-D data sets	171 s	46421	512 MB
<i>FFTPDE</i>	FFT solver for 3-D PDEs	5537 s	—	224 MB
<i>MGRID</i>	multigrid solver for 3-D potential	913 s	—	431 MB
<i>MATVEC</i>	Matrix-vector multiplication	1006 s	—	385 MB
<i>Sphinx</i>	Off-line speech recognition	72 s	66358	181 MB

Table 1: Benchmark characteristics. Run time is on an unmodified kernel, with no speculative execution.

4 In-kernel speculative execution

In this section we present and evaluate our basic design for leveraging speculative execution within the operating system. Our design is similar in spirit to the previous user-level design, but has the advantages of being much easier to implement and more accessible to users. Moreover, we demonstrate that not only does this design provide large benefits similar to those of the prior system for explicit-I/O applications, but also it can provide substantial benefits for swapping applications.

4.1 Basic design

We add a new type of process to the system – a *speculative* process. A speculative process is created by forking a normal process the first time it blocks on a disk request. A speculative process is completely destroyed only when its parent process exits. The operating system treats speculative processes differently from normal processes only in order to: 1) ensure that speculative execution is *safe* (that is, speculative processes do not produce output or otherwise change the results of executing applications); 2) enable speculative processes to issue prefetches on behalf of their parent processes; and 3) restrict the resource utilization of speculative processes so that they cannot hurt the performance of normal processes.

Notice that speculative processes are never created for normal processes that perform no disk I/O. We also allow users to opt out of speculative execution by setting a specific environment variable.

4.1.1 Safety

It is easy to ensure that speculative execution is safe because operating systems already severely restrict the ways in which different processes can affect one another. As a result, a system needs to restrict speculative processes in only three simple ways to ensure safety.

First, on most systems, a forked process shares file pointers with its parent process, and inherits write access

to mapped files and shared memory segments. To ensure safety, when forking speculative processes, file pointers are instead copied, and write access to mapped files and shared memory segments is changed to copy-on-write access.

Second, systems typically establish a relationship between a parent and child process such that information about the child process is propagated to its parent. For example, when a child process is destroyed, the operating system typically delivers a signal to its parent. To ensure safety, we sever these ordinary relationships between speculative processes and their parents.

Finally, the system must restrict which system calls speculative processes can perform. For example, a speculative process must not be allowed to modify the file system or send signals to normal processes. Therefore, speculative processes are required to perform access checks before proceeding with system calls. If the requested system call is not safe (or *may* not be safe, since implementations should be conservative), the access checks either cause the speculative thread to return immediately, or alter the requested system call so that it is safe. For example, requests to map file regions with write access are converted to requests to map those file regions privately with copy-on-write semantics. It is easy to force speculative processes to return immediately from any unsafe system call as most operating systems contain a single access point for all system calls, such as a software trap handler. We modified this trap handler such that speculative processes perform a table lookup indexed by the system call number and then, depending on the value encoded in the table, either return immediately or continue with the system call. Altering system calls issued by speculative processes requires slightly more effort in the form of individually modifying the call-specific handler for each such call. However, this effort was required for only a small number of calls, such as `mmap` and “multiplexed” command interfaces such as `ioctl`.

4.1.2 Prefetching

A speculative process generates prefetch requests on behalf of its parent process by initiating a *non-blocking* prefetch whenever a normal process would have blocked on a disk read. Speculative execution then proceeds without the non-resident data. If the disk read resulted from an explicit file read call, any memory-resident data specified by the call is copied into the specified user buffer while regions of the buffer that would ordinarily be filled by non-resident data are unchanged. This allows the speculative process to make use of all available data during its subsequent execution. If the disk read resulted from a page fault then the system, as usual, allocates a page frame, updates the appropriate page table entry to map the new page frame, and initiates a disk read of the appropriate data. However, rather than blocking until the disk read completes, the speculative process returns from the fault immediately so that it can run ahead of its parent process. Since its page table has been updated, any subsequent accesses it makes to the same page will not generate another page fault.

It is possible that a speculative process will execute, but not succeed at generating accurate prefetches for its parent. This may occur because future execution depends on non-resident data, a system call that speculative processes are not allowed to perform, or inter-process communication through shared memory (since, as discussed in the prior section, shared pages are mapped copy-on-write in speculative processes). To increase the chances that the speculative process will generate useful prefetches, whenever the parent process is about to block waiting for some data for which its speculative child failed to generate a prefetch, it attempts to *synchronize* its speculative child, where synchronizing is just like forking except that we reuse the speculative child's process structure. A parent process can easily detect when its speculative child failed to generate a prefetch because a process must first allocate a page frame to hold data before issuing a disk read for that data. Therefore, if no page frame has been allocated for some data, then no prefetch has been issued for that data. We attempt to hide the observed cost of synchronization within the time to fetch the data from disk, during which the parent process would ordinarily block, by issuing the disk request before we begin synchronizing.

On a typical system, a process initiates file readahead while servicing a file read system call, and initiates page cluster reads when it faults on a page that is neither in memory nor already being fetched from disk. However, if a speculative process is issuing the correct prefetches, then the default prefetch heuristics are at best redundant and might waste memory and disk bandwidth by prefetching unneeded data. We therefore disable these

heuristics if a speculative process has prefetched the data that its parent process is requesting.

4.1.3 Basic resource control

Supporting speculative execution consumes processing cycles, disk bandwidth, and memory. Ideally, speculative execution only uses resources that would otherwise be wasted, so it cannot hurt system performance. This section describes how we restrict the processing time of speculative processes, and the disk bandwidth and memory of speculative prefetches.

Processor cycles. We ensure that speculative execution does not steal processing cycles from normal processes by only scheduling speculative processes when nothing else is runnable. Furthermore, speculative processes are preempted as soon as a normal process becomes runnable. Rather than relying on the operating system's existing priority mechanism, which does not guarantee the desired behavior, we implement this with a simple modification (six additional lines) in the kernel scheduler code.

Prefetching resources. A speculative prefetch can steal disk bandwidth from normal processes by delaying a disk request from a normal process. Moreover, premature prefetching can hurt performance, even if all the prefetches are accurate, by causing useful data to be unnecessarily ejected from memory. As suggested by Patterson, et. al. [5], we limit the maximum delay of a normal request by only issuing a prefetch to a disk which has no more than one request outstanding. We also use their idea of a *prefetch horizon* – a system-dependent, calculable maximum number of prefetches in advance beyond which there is unlikely to be a benefit to initiating a prefetch – to throttle speculative processes that are generating prefetches too quickly. If a speculative process attempts to prefetch further ahead than the prefetch horizon, it is marked non-runnable until its parent process either accesses some of the data it prefetched, or synchronizes it.

Finally, to avoid wasting resources, if a speculative process begins executing process termination code (e.g. as a result of issuing an `exit` system call or generating a terminating exception), then the operating system checks whether its parent process is terminating. If not, then the speculative process is not allowed to terminate; instead, its memory is reclaimed and it is marked non-runnable until its parent process synchronizes it.

4.2 Performance of the basic design

Figure 2 shows the overall performance of the basic design. For all four explicit-I/O applications, shown in the leftmost section, our in-kernel design delivers large

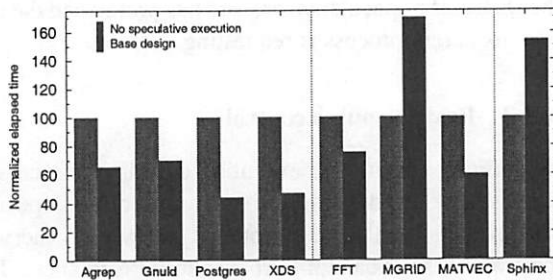


Fig. 2: Performance of our basic in-kernel design.

improvements, reducing elapsed times by 35% to 56%. These benefits are comparable to those delivered by the user-level design [3], and are achieved for the same reasons. In particular, unlike file readahead, the speculative execution approach enables prefetching across files and can leverage the decision paths encoded in applications to generate accurate prefetches for accesses that are seemingly random.

The results for swapping applications, shown in the central section, and our combination application, Sphinx, are more varied. We deliver substantial benefits for FFTPDE and MATVEC, but we degrade the performance of MGRID and Sphinx. Thus, our base design performs poorly compared with the compiler-based approach [4], which delivered a substantial performance benefit for MGRID.

Table 2 provides more detailed information about the executions. Unsurprisingly given the overall results, for all the applications except MGRID and Sphinx, speculative execution significantly reduces both the number of I/O stalls and the I/O stall time experienced by normal execution.

One potential concern with speculative execution is that it will not generate prefetches early enough to hide a substantial amount of I/O stall time. The figures for full speculative prefetches show that the vast majority of speculative prefetches actually complete before the data is accessed during normal execution; in other words, there are very few partial stalls on in-progress prefetches. Another potential concern is that, as with any heuristic approach, speculative execution may generate prefetches for data that will not be used, wasting both memory and disk bandwidth. The figures for unused speculative prefetches show that speculative execution is perfectly accurate for Agrep and MATVEC. For the other benchmarks, speculative execution generates some needless prefetches, but is always much more accurate than the operating system's default file readahead and page cluster heuristics. Furthermore, because good speculative prefetches disable the operating system's default prefetching heuristics, we are able to avoid a large

proportion of needless prefetches for all benchmarks except Sphinx. This further helps performance by reducing contention for memory and disk bandwidth.

On the other hand, comparing the figures for explicit-I/O and swapping applications reveals that synchronization is substantially more expensive for swapping applications. In particular, comparing the synchronization times to original execution times (shown in the first column) reveals that MGRID is synchronizing for almost half of its original execution time. This suggests one way in which the base design is inefficient for swapping applications, and ineffective for MGRID in particular. For many applications, we also notice a substantial increase in the number of copy-on-write faults. Finally, Sphinx demonstrates a different potential problem with speculative execution. The memory use of speculative execution can cause useful data to be prematurely ejected from memory. This is revealed by how speculative execution *increases* the total number of I/O stalls for Sphinx. We address these weaknesses of the basic design in the next two sections.

5 Improved prefetching for swapping applications

Although the basic design discussed in the previous section works well for explicit-I/O applications, it can hurt the performance of applications that exploit virtual memory. In this section, we discuss two simple additions to standard operating system mechanisms which can greatly improve prefetching performance for memory-intensive applications.

5.1 Fast, preemptible refork

Since they rely on file- and swap-backed virtual memory to hide I/O, swapping applications typically have very large page tables. This presents two problems. First, when synchronizing its speculative child, a parent process can be substantially delayed by the time required to release its speculative child's old state and then make a fresh copy of the parent process's state. Second, the cycles consumed in synchronization reduce the number of spare processing cycles in which speculative execution can make progress. We reduce the cost of synchronization by adding a fast, preemptible refork operation.

Recall (from Section 4.1.2) that the parent process begins synchronizing its speculative child only after issuing a disk request that would ordinarily cause it to block. If the synchronization operation does not complete before the disk request completes, then the parent process will no longer need to block after synchronizing its child.

Benchmark	Specx enabled?	I/O Stalls		Spec prefetches		Unused prefetches		Sync delay	CoW faults
		Total	Time	Total	Full	Spec	Readahead		
<i>Agrep</i> (42s)	No	14080	30s	–	–	–	50	–	286
	Yes	4239	17s	7350	92%	0	50	0s	326
<i>Gnuld</i> (30s)	No	5434	26s	–	–	–	5245	–	8
	Yes	3228	15s	4684	74%	641	472	2s	8
<i>PostgreSQL</i> (9915s)	No	1056013	8471s	–	–	–	823234	–	259
	Yes	267498	3230s	1201033	87%	25101	43011	45s	2714
<i>XDS</i> (171s)	No	22887	159s	–	–	–	178430	–	13
	Yes	5905	63s	44512	87%	634	1632	7s	11376
<i>FFTPDE</i> (5537s)	No	439221	5345s	–	–	–	3188757	–	8
	Yes	201384	3024s	1384548	86%	583	116134	929s	391281
<i>MGRID</i> (913s)	No	87594	757s	–	–	–	130413	–	8
	Yes	59732	1020s	1095623	90%	14	21390	428s	179714
<i>MATVEC</i> (1006s)	No	105518	925s	–	–	–	332361	–	7
	Yes	36114	375s	893238	80%	0	12788	75s	91
<i>Sphinx</i> (72s)	No	6238	43s	–	–	–	19711	–	1386
	Yes	6957	58s	8417	68%	909	20831	6s	1388

Table 2: Effectiveness of prefetching by the basic design, compared with a system which performs no speculative execution. I/O stalls is the number of, and elapsed time during, I/O stalls experienced by normal execution. Speculative prefetches is the number of speculative prefetches, and the percentage of those prefetches that completed before the data was requested by normal execution. Unused speculative prefetches is the number of speculative prefetches that prefetched data that was not used before being ejected from memory. Unused readahead prefetches is the same statistic for prefetches generated by the operating system's file readahead and page clustering heuristics. Sync delay is the total time used to synchronize the speculative child process, some of which is hidden by disk access latency. CoW faults is the total number of copy-on-write faults experienced during normal execution.

Since the speculative process is not allowed to steal cycles from non-speculative processes, this means that the speculative process will not be able to run ahead of the parent process. (At best, on a multiprocessor, it may run in tandem with its parent). Therefore, there is no benefit to requiring that the parent complete a synchronization, and we are better off ensuring that synchronization does not needlessly delay a parent process. We accomplish this by periodically checking whether the disk request has completed. If the read has completed, the parent simply stops its synchronization attempt, and the speculative child continues to be non-runnable. The parent will attempt to complete a synchronization the next time it is delayed by disk I/O.

While this usually hides the cost of synchronization from the parent, it may *increase* the synchronization delay perceived by the child. However, we observe that the parent process will attempt to synchronize its child every time it needs to access any data that is not in memory. Swapping applications, which typically have a large working set, will therefore synchronize quite often, and so the page tables are unlikely to have changed significantly. This allows us to reduce synchronization time by releasing and updating only those page table entries that have changed in the parent or the speculative child since the last synchronization. Moreover, this optimization complements preemptible reforking; if the parent

cannot complete a synchronization while one of its disk requests is being serviced, the partial synchronization is likely to reduce the amount of work it must perform to complete the synchronization the next time it is delayed by disk I/O.

5.2 One-way copy-on-write

If synchronization follows the usual forking semantics, the parent process will experience a copy-on-write page fault for each page that it attempts to modify before its speculative child. Copy-on-write faults can introduce substantial delay because of the page allocation and data copy that are required. In particular, although the cost of page allocation is generally quite low, it can increase dramatically when memory contention is high [2]. Observing that safety only requires that a page be copied if the child attempts to modify it, we avoid adding page allocations to parent processes by adding support for one-way copy-on-write; that is, we allow normal processes to make modifications which may be observed by speculative process.

Supporting one-way copy-on-write requires only a few modifications. First, while synchronizing, only the speculative child's page table entries and memory region mappings are marked as copy-on-write. We also add a speculative reference count on page frames and

Benchmark	Refork type	Refork time		Refork attempts	
		Total	Mean	Total	Completed (%)
<i>FFTPDE</i>	Normal	929 s	7 ms	136645	136645 (100%)
	Fast	274 s	2 ms	118581	105879 (89%)
<i>MGRID</i>	Normal	428 s	12 ms	37065	37065 (100%)
	Fast	144 s	3 ms	48668	39583 (81%)
<i>MATVEC</i>	Normal	178 s	10 ms	17669	17669 (100%)
	Fast	75 s	3 ms	24951	17083 (68%)
<i>Sphinx</i>	Normal	13 s	3 ms	3769	3769 (100%)
	Fast	6 s	2 ms	2465	2254 (91%)

Table 3: Synchronization cost: the effect of a fast, preemptible re-fork.

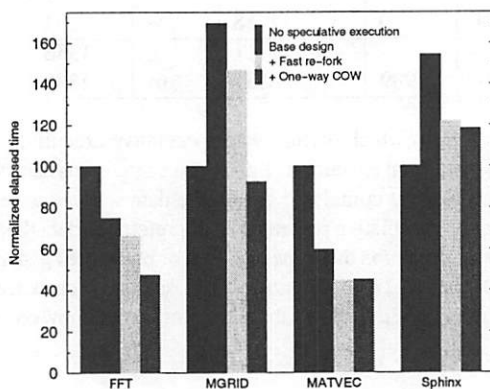


Fig. 3: Performance of speculative prefetching with fast re-forking and one-way copy-on-write.

swap slots, which track the number of references from speculative processes. When servicing a page fault during normal execution for a resident swap-backed page, we subtract the speculative reference count from the total count to determine whether the page must be copied. Finally, whenever a speculative process services a copy-on-write fault, we decrement the reference count on the original page.

5.3 Swapping application performance

Fast, preemptible reforking and one-way copy-on-write have negligible effect (less than one percent) on the execution times of our explicit-I/O applications, which use a modest amount of virtual memory. Figure 3 shows the degree by which our swapping applications benefit from these mechanisms. All four swapping applications run significantly faster compared with our baseline approach, with the speedup across applications approximately equally attributed to the two mechanisms. Moreover, while MGRID runs 60% slower with baseline speculative prefetching, fast, preemptible reforking and one-way copy-on-write eliminate this overhead.

Benchmark	CoW faults		
	No Specx	Basic	One-way
<i>FFTPDE</i>	8	391281	8
<i>MGRID</i>	8	179714	8
<i>MATVEC</i>	7	91	7
<i>Sphinx</i>	1386	1388	1387

Table 4: The effect of one-way copy-on-write on the number of copy-on-write faults experienced during normal execution. Basic is our base speculative execution design. One-way includes the one-way copy-on-write optimization. No Specx has all speculative execution disabled.

Detailed information about the benefits of fast, preemptible reforking are presented in Table 3. For the three scientific applications, the improvement over our baseline approach is dramatic: total refork time is reduced by a factor of three or more. Importantly, unlike normal reforks, the average fast refork time is much shorter than an average disk access for all of the benchmarks. Faster reforking enables a high proportion of these preemptible refork attempts to complete, and provides speculative execution with more time to run before it is preempted by normal execution.

Fast, preemptible reforking can increase the number of synchronization attempts (as with both MGRID and MATVEC) because preempted attempts will quickly be retried, the next time normal execution stalls. This mechanism increases the number of *completed* synchronizations, however, only for MGRID. In terms of execution time, this increase is far outweighed by the reduced refork time.

Examination of detailed application traces reveal that the improved performance of Sphinx is due to reforks being preemptible. Not only does this prevent normal execution from being needlessly delayed, but also it reduces the time during which the speculative process is runnable. Because Sphinx has a large memory footprint, leaving speculative execution non-runnable can substantially reduce memory contention, which is the main reason for degraded performance during this benchmark.

Table 4 shows that the one-way copy-on-write mechanism delivers dramatic reductions in the number of copy-on-write faults during normal execution for FFT-PDE and MGRID. The performance benefit of these reductions can be seen from the results in Figure 3. These performance improvements are due not only to the direct benefit of fewer copy-on-write faults, but also to the indirect benefit of decreasing memory contention by requiring fewer page allocations. MATVEC and Sphinx gain no noticeable benefit from one-way copy-on-write because, even with this mechanism disabled, normal execution experiences very few copy-on-write faults due to speculative execution. (Each benchmark experiences some unavoidable number of copy-on-write faults as a result of write accesses within shared libraries, which can be seen from the count of copy-on-write faults when speculative execution is disabled.)

6 Controlling memory overhead

The simple resource control mechanisms in the basic design (Section 4.1.3) are sufficient for a system that always has abundant memory. Most systems, however, are not so over-supplied. Further, it seems likely that the performance of memory-intensive applications would be harmed by ineffective speculative execution. In this section, we describe our mechanism for controlling the memory overhead incurred by speculative processes. This mechanism enables practical speculative execution on systems which may experience memory contention.

It is difficult to control memory overhead while still enabling effective speculative execution because, unlike processor or disk bandwidth, it is usually not possible to determine whether the resource is actually being wasted, and can therefore be used without hurting system performance. For example, even if the memory mapped by all extant processes is much less than the total amount of memory in the system, there is always a chance that a page that contains file data will be re-accessed.

The previous user-level design relied on the TIP prefetching and caching manager [5]. TIP performs *cost-benefit analysis* to determine when allocating some memory for prefetching would be more beneficial than allowing the LRU file cache to retain that memory. TIP is not a complete memory management solution for speculative execution, however, because speculative processes need to allocate memory not only to hold prefetched data, but also to dynamically allocate memory and make modifiable copies of pages to which they have copy-on-write access. Furthermore, the benefit of allowing a speculative process to allocate memory for its own use depends entirely on whether it will subse-

quently be able to issue useful prefetches, which depends on factors not within its control, such as how often it will be preempted.

We preserve the principle idea of a cost-benefit framework but, rather than building a complicated system model to predict the benefit of each speculative allocation request, we propose a simpler *reactive* approach to controlling memory overhead. Specifically, we estimate the benefit that a speculative process has already provided by prefetching data, and the cost it has already incurred by its memory consumption. This allows us to estimate the current benefit or cost of each speculative process and, by disabling processes accordingly, restrict the overhead incurred by ineffective speculative execution.

6.1 Reactive cost-benefit analysis

We estimate the cost and benefit of each speculative process with the assistance of an *eviction list* (or *ghost buffer* [8]). The eviction list tracks the non-resident pages that would most likely be in memory if no speculative execution had taken place. Each list entry uniquely identifies the disk block of one such page. To determine how many entries the list should contain, we track the number of *speculative pages* that are in memory only because of speculative execution. The eviction list is a FIFO to which we add an entry for each non-speculative page that is evicted from memory, and remove an entry if the FIFO is already full when a new page is added.

The eviction list allows the system to estimate the number of I/O stalls that speculative executions added to, or removed from, normal executions. In particular, when a normal process requests a disk read, if there is a matching entry in the eviction list, then we have identified an *added stall* that would not have occurred in the absence of speculative execution. Conversely, when a normal process avoids a disk read by using a speculative page, we have identified a *removed stall* that was prevented by speculative execution. The eviction list further aids accurate identification of removed stalls by allowing the system to detect when a speculative prefetch is merely refetching previously-resident data that was only evicted because of speculative memory use. Figure 4 describes the updates and accesses that can occur to the eviction list and associated performance counts in greater detail.

With the resulting per-process counts of removed stalls (*RemovedStalls_p*), and the system-wide count of added stalls (*TotAddedStalls*), we could estimate the net overhead of each speculative process as the overall cost/benefit for its execution so far. We could then use this measure to disable a speculative process whenever its net estimated overhead is above some selected

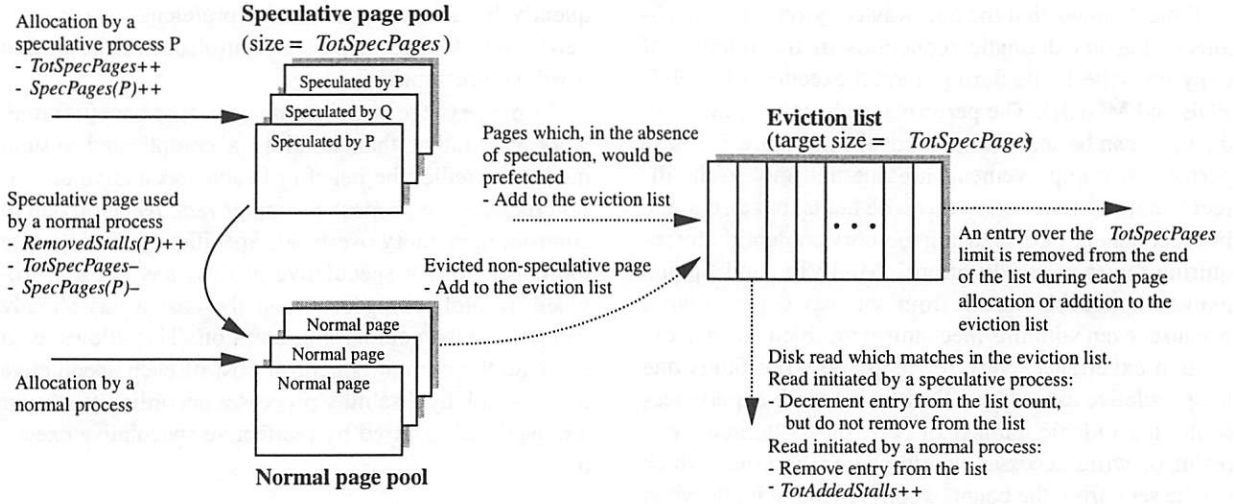


Fig. 4: Using the eviction list to calculate added and removed stalls. Many page allocations and evictions update or query the eviction list. This diagram enumerates the possible transitions that can occur. Dotted lines represent page identifiers being added to or removed from the eviction list.

threshold. Unfortunately, this approach would be unresponsive to changes in the effectiveness of a speculative process. Such changes might be caused by different phases within the application which affect prefetching accuracy, or simply by varying memory contention from concurrent processes. We therefore use an approach that not only bounds the estimated overhead of enabled speculative processes, but also is responsive to changes in the estimated overhead of speculative processes.

The remainder of this section describes the *reactive* approach that we implemented and evaluated. Many other possibilities exist, but a survey is beyond the scope of this paper.

We divide system time into periodic *intervals*, where t denotes the current interval. At the end of each interval, a pair of cost-benefit estimates are updated for each speculative process based on the estimates and accumulated stall counts from the previous interval, and whether the speculative process was enabled or disabled during that interval. We use these estimates (and any accumulated stall counts since the last interval) to estimate the overhead at any time, in order to decide when to enable or disable each speculative process, as follows.

If a speculative process was *enabled* (i.e. marked runnable), we exponentially decay the previous stall estimates at the end of each interval:

$$\begin{aligned} Cost_p(t) &= (1 - \alpha) \times Cost_p(t - 1) + \alpha \times AddedStalls_p(t - 1) \\ Benefit_p(t) &= (1 - \alpha) \times Benefit_p(t - 1) + \alpha \times RemovedStalls_p(t - 1) \end{aligned}$$

Per-process counts of removed stalls are maintained directly by observing when speculative pages are first referenced by a normal process. However, to calculate per-process *added* stalls, the system-wide total for each in-

terval is divided among the speculative processes in proportion to their memory use:

$$AddedStalls_p(t) = TotAddedStalls(t) \times \frac{SpecPages_p}{TotSpecPages}$$

We then combine the cost-benefit estimates in the simplest manner to obtain the *overhead* of each enabled speculative process:

$$Overhead_p(t) = Cost_p(t) - Benefit_p(t)$$

If a speculative process's estimated overhead is non-negligible, we mark it non-runnable and reclaim its memory. Therefore, if memory is tight, speculative processes that consume more memory will need to prefetch more effectively to remain runnable.

If a speculative process was *disabled* (i.e. marked non-runnable), we simply add to the previous cost and benefit estimates at the end of each interval:

$$\begin{aligned} Cost_p(t) &= Cost_p(t - 1) + AddedStalls_p(t - 1) \\ Benefit_p(t) &= Benefit_p(t - 1) + RemovedStalls_p(t - 1) \end{aligned}$$

Our overhead calculation then includes the period over which the process has been stopped. Let s be the interval during which non-runnable process p was last executed:

$$Overhead_p(t) = \frac{Cost_p(t) - Benefit_p(t)}{t - s}$$

Thus we decay the overhead estimate over time, which allows the system to set an upper bound on the net estimated overhead before a speculative process should be

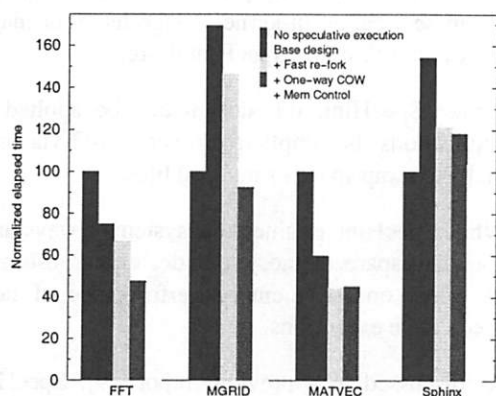


Fig. 5: Overall benefit of memory control for swapping applications (with the default machine configuration).

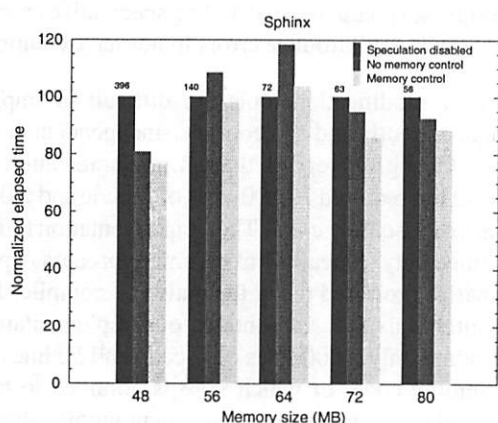


Fig. 6: The effect of memory control on the Sphinx benchmark, for a range of memory configurations. Run times for each memory size are normalized to the execution time with no speculation. These times are shown above the corresponding bar, in seconds.

allowed to run again. Notice that this property is independent of the manner in which the stall counts are decayed for enabled speculative processes.

There is some scope for investigating more informed techniques for reenabling speculative processes; for example, we could allow a speculative process to continue running and issuing prefetch requests, but prevent these requests from reaching the disk or allocating memory. This would reduce resource pressure compared with a fully-enabled speculative process, while providing us with more information to help determine when it would be worthwhile to reenable speculative execution. However, a major benefit of our current approach is its *conservatism* — a more informed approach would risk a harmful increase in resource pressure for just those applications that require most care.

Mem size	Spec	M	Threshold crossings	Syncs	Stall time
48 MB	Off	-	-	-	315s
	On	N	-	18303	224s
	On	Y	15	14790	218s
56 MB	Off	-	-	-	106s
	On	N	-	8356	102s
	On	Y	12	1963	101s
64 MB	Off	-	-	-	43s
	On	N	-	3337	47s
	On	Y	10	1979	41s
72 MB	Off	-	-	-	34s
	On	N	-	2282	25s
	On	Y	4	957	23s
80 MB	Off	-	-	-	28s
	On	N	-	1955	21s
	On	Y	5	729	20s

Table 5: The effect of memory control on synchronization attempts (Syncs) and I/O stall time. Three sets of results are shown for each memory configurations: speculation disabled, speculation enabled, and speculation with memory control (M) enabled. Threshold crossings is the number of times speculative execution is disabled due to intolerable memory overhead. Execution times without speculative execution are shown in Figure 6.

6.2 Evaluation of memory control

Figure 5 shows the performance benefit of memory control for our swapping applications. For our benchmarks, memory control only has a significant effect for Sphinx. This indicates that, as desired, memory control does not diminish the performance benefit of speculative execution when speculative execution is effective.

Figure 6 shows Sphinx results when the system is configured to have differing amounts of usable memory. In all memory configurations, our mechanism provides some benefit to Sphinx as compared to having no memory control mechanism. Moreover, in 56MB and 64MB configurations, memory control is able to eliminate, or significantly reduce, the performance penalty that occurs without memory control. These results suggest that this mechanism can effectively prevent speculative execution from harming performance in cases where speculative prefetching has too few resources to provide benefit. This is an important requirement if a prefetching system is to be deployed ubiquitously in a system.

From the detailed information listed in Table 5, it is evident that much of the performance benefit comes from reduced stall time. However, further gains are possible due to reduced memory contention in the absence of speculative execution. This will, for example, reduce the number of soft page faults experienced by normal execution for pages which are in the process of being

laundered, and can also reduce the cost of page allocations.

However, the adverse effects of speculative execution can linger for some time after it has been disabled, as the resulting 'gap' in memory cannot be immediately filled with useful data. Furthermore, in estimating overhead, our mechanism currently assumes average stall times. Actual stall times can vary greatly, which is why memory control does not entirely eliminate the penalty of speculative execution in our 64MB results.

Our mechanism disables speculative execution quite infrequently. However, the number of synchronization attempts is considerably reduced, indicating that, when speculation is disabled, it remains disabled for a considerable period of time. This is due to our conservative algorithm, which ensures that speculation is only reenabled when its net estimated overhead is negligible.

Surprisingly, speculative execution improves application performance on a 48MB system without the benefit of the memory control mechanism. This is due to more accurate prefetching compared with the default read-ahead heuristic; the memory cost of speculative execution is more than offset by the reduction in needlessly prefetched data. However, even in this case, memory control still provides a further gain of nearly 10% by disabling speculative execution while it is less effective. In the 72MB and 80MB results, basic speculative execution has enough memory available to provide overall benefit; however, the control mechanism is still able to identify a handful of places where it is beneficial to temporarily disable speculative execution.

7 Comparison to previous design

This paper describes an in-kernel design for applying the speculative execution approach to arbitrary, unmodified executables. In contrast, the previous design [3, 9], called *SpecHint*, specifies an automatable procedure for modifying application binaries to apply the speculative execution approach.

The SpecHint design has two advantages compared with an in-kernel design. First, SpecHint requires no operating system support specific to speculative execution, allowing deployment on systems where OS modifications are not feasible. Second, SpecHint can exploit static analyses and transformations to specialize the application code for speculative execution. For example, calls to expensive library functions, such as `printf`, can be removed to speed up speculative execution. More complex analysis might remove loops with data-dependent bounds which could trap speculative execution and prevent it from generating further I/O hints.

However, in addition to increasing the accessibility

of speculative execution by providing it as an operating system service, an in-kernel design has four major advantages relative to the SpecHint design.

1. Unlike SpecHint, our design can be applied to applications that implicitly generate I/O via page faults to swap space or mapped files.
2. While SpecHint assumed that systems always have abundant spare memory, our design can estimate its effect on the memory performance of non-speculative executions.
3. As discussed in a previous report [9], SpecHint makes some assumptions in its attempts to ensure that the modified binary will not produce different results than the original binary. While these assumptions will hold in most cases, an in-kernel design can guarantee that adding speculative executions will not introduce errors in normal execution.
4. Binary modification tools are difficult to implement correctly and in a compiler-independent manner. Chang [9] reports that an implementation of SpecHint required 19,000 lines of C code and 6,000 lines of assembly code. That implementation transformed only statically-linked, single-process Alpha binaries produced using the native `cc` compiler for Digital Unix 3.2. In contrast, our implementation required only 5,000 lines of C code and 50 lines of assembly code, of which 90% is confined to two new files and memory management modifications, and it can be applied to arbitrary Linux x86 executables. Moreover, since our implementation contains fewer than 100 lines of x86-specific code, it should easily extend to handle arbitrary Linux executables on other platforms as well.

8 Related and future work

Common access pattern heuristics [10, 11, 12, 13] prefetch according to a small set of access patterns that occur frequently, such as sequential access. The system checks whether recent accesses fit a known pattern and, if so, issues prefetches by extrapolating the pattern. The simplicity of these approaches is an advantage. However, if accesses do not fit a known pattern then the heuristic cannot help, and may even hurt, application performance.

Dynamic history-based approaches [14, 15, 16, 17, 18, 19] initiate prefetching based on patterns inferred from previous access sequences. Although these systems can discover new patterns and exploit this knowledge across multiple applications, they cannot help with

non-repetitive accesses and may need to observe a large number of data accesses before an accurate pattern can be inferred. Furthermore, the history data can itself occupy a large amount of memory.

In *static analysis-based* approaches [20, 4], a compiler analyzes the application to deduce the accesses it will make during execution. It then inserts hints into the application to inform a run-time prefetcher. Although these approaches have low run-time overhead, the required interprocedural analysis is difficult. As a result, existing systems benefit only looping array codes. Furthermore, system-wide deployment will require all applications to be recompiled.

Cao *et al.* [21] note that aggressive prefetching strategies can do more harm than good by evicting data from memory which is later accessed. They propose a set of rules that an integrated prefetching and caching policy should follow to avoid hurting performance, and suggest two conforming policies. However, these policies assume perfect knowledge of application reference streams. Later work [22] mentions, but does not evaluate, possible heuristics for accommodating imperfect reference streams.

The TIP prefetching system [5] uses a cost-benefit model to control prefetching into, and eviction from, a file cache of limited size. When an application provides an access hint, the possible *benefit* of prefetching that block is weighed against the *cost* of evicting the least valuable block in the cache. However, because TIP only deals with buffer allocation for prefetched data, the cost-benefit analysis is simpler than ours, which must also deal with page allocations of no direct benefit to normal execution.

One aspect of our design that we would like to improve further is memory management. Demke and Mowry [2] demonstrate substantial benefits by proactively evicting pages from memory when memory contention is high. They describe a compile-time technique which deduces the accesses an application will make during execution and inserts *release hints* for blocks that are unlikely to be accessed in the near future. When coupled with their compile-time prefetch hints, a run-time system can approximate Cao's scheduling rules. Unfortunately, static insertion of release hints is limited by difficult interprocedural analysis. We hypothesize that a speculative execution approach may be able to expand the range of applications for which release hints could be automatically generated.

Finally, Chang [9] demonstrates that speculative execution can improve system performance even when concurrent processes contend for processing cycles or disk bandwidth. While our memory control mechanism was designed with multi-process systems in mind (for example, speculative process overheads are measured individ-

ually) it has yet to be evaluated in a multi-programming environment.

9 Conclusions

Recent work demonstrated that speculative execution has the potential to greatly improve the performance of I/O-intensive applications, while overcoming the limitations of compiler-assisted prefetching approaches. However, the previous user-level design requires the implementation of a complex, architecture-specific binary modification tool, benefits only explicit-I/O applications that have been transformed by such a tool, and does not limit the overhead incurred by increased memory contention.

In this paper, we present an in-kernel design for capturing the benefits of speculative execution while overcoming these limitations. We demonstrate that, for explicit-I/O applications, a simple design which leverages existing operating system mechanisms delivers benefits comparable to the prior user-level design. We then show that specialized versions of standard OS mechanisms – a fast, preemptible reforking operation and directional copy-on-write – greatly increase the benefits provided to swapping applications. Finally, we demonstrate a mechanism for limiting speculative overhead, while not impeding beneficial speculative execution, by scheduling speculative executions based on their memory impact. Our experience in implementing and evaluating speculative execution within Linux suggests that providing speculative execution within an operating system can be both feasible and effective.

Acknowledgements

We would like to thank Garth Gibson, Khalil Amiri and the anonymous reviewers for providing many helpful comments that greatly improved the paper. We would also like to thank Angela Demke for making her I/O benchmark suite available to us.

References

- [1] E. Grochowski. IBM magnetic hard disk drive technology. <http://www.almaden.ibm.com/sst/html/leadership/leadership.htm>, 2001.
- [2] A. D. Brown and T. C. Mowry. Taming the memory hogs: Using compiler inserted releases to manage physical memory intelligently. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 2000.

- [3] F. Chang and G. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.
- [4] T. Mowry, A. D. Brown, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, 1996.
- [5] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.
- [6] D. Bitton, C. Orji, and C. Turbyfill. The AS3AP benchmark. In *Database and Transaction Processing Sys. Performance Handbook*. Morgan Kaufmann, 1991.
- [7] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, 1991.
- [8] M. Ebling, L. Mummert, and D. Steere. Overcoming the network bottleneck in mobile computing. In *Workshop on Mobile Computing Systems and Applications*, 1994.
- [9] F. Chang. Using speculative execution to automatically hide I/O latency. Technical Report CMU-CS-01-172, Carnegie Mellon University, 2001.
- [10] R. J. Feiertag and E. I. Organisk. The multics input/output system. In *Proceedings of the 3rd ACM Symposium on Operating Systems Principles*, 1971.
- [11] M.K. McKusick, W.J. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.
- [12] D. Kotz and C. Ellis. Practical prefetching techniques for parallel file systems. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems (PDIS)*, 1991.
- [13] T. Madhyastha, G. A. Gibson, and C. Faloutsos. Informed prefetching of collective I/O requests. In *Proceedings of the ACM/IEEE SC99 Conference*, 1999.
- [14] C. Tait and D. Duchamp. Detection and exploitation of file working sets. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991.
- [15] M. L. Palmer and S.B. Zdonik. FIDO: A cache that learns to fetch. In *Proceedings of the Conference on Very Large Data Bases*, 1991.
- [16] K.M. Curewitz, P. Krishnan, and J.S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM Conference on Management of Data (SIGMOD)*, 1993.
- [17] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer Technical Conference*, 1994.
- [18] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of the USENIX Winter Technical Conference*, 1997.
- [19] T. Kroeger and D. Long. The case for efficient file access pattern modeling. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HOTOS)*, 1999.
- [20] K. S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, 26(10):938–947, 1977.
- [21] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1995.
- [22] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.